An Experiment in the Design of Modular Expert Systems with Special Reference to Fault Diagnosis

James P.H. Coleman

Submitted in fulfillment

of the requirements for the degree of

Master of Science

Department of Electrical Engineering

and Computer Science

University of Tasmania

October 1989

Except as stated herein, this thesis contains no material which has been accepted for the award of any other degree or diploma in any university. To the best of my knowledge and belief, this thesis contains no copy or paraphrase of material previously published or written by another person, except where due reference has been made in the text of the thesis.

James P H Coleman

Acknowledgement
I wish to acknowledge the assistance of my supervisor, Phil Collier for his time and guidance in my research and when writing up this thesis. I would also like to thank my father and Ed Kazmierczak for their help with the writing up, my family and the G.C.S.B who provided much inspiration on our field research trips.

ABSTRACT

A method for dealing with three of the major problems in hybrid rule-based expert systems is proposed. They are; the problem of managing the complexity of the knowledge base, the problem of graceful degradation and the problem of the logics used to manipulate the rules, which do not fit easily with human expertise. An experimental expert system, Aristotle, is developed in which modules are used as a tool to reduce complexity. Modules group related rules together into components, which interact with the outside world through parameters. The module also provides a suitably sized object on which extra (partial) knowledge can be attached. It is proposed to use partial knowledge to reason, at a general level, about components and groups of components, giving the appearance of a knowledge base, which degrades gracefully. A five-value logic is introduced with the extra logic values *irrelevant*, *do-not-know* and *unknown*. This logic is able to reason about unusual situations without having to explicitly check for them. This thesis demonstrates, that modules, partial rules and the five-value logic, can overcome, to an extent, the problems mentioned earlier.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION

Expert System development began in the late 60s with DENDRAL, which was built at Stanford University, initially using conventional programming techniques, but after two years of work, the group changed strategy and started to use production rules. They reasoned that Rules would "...facilitate rapid adjustment to alternative hypotheses from the users" (Little 1986). DENDRAL was designed to identify chemical compounds based on the their infra-red spectra. It was a successful project and is still in use today.

Stanford's next expert system was MYCIN, which made production rules one of the most popular techniques for representing knowledge, and for the next decade was the archetypical expert system. It is still famous for its use of rules.

Since then, expert systems have been recognised not only as a subject for research, but as having practical uses in the commercial world. Systems have been developed to provide advice on a range of topics from diagnosis (medical, fault etc) to planning and monitoring; and many businesses are now spending large sums of money to develop expert systems for their particular applications. They are expecting the pay-offs from expert systems to come in the following ways: (Hewett 1986)

- Increased productivity,
- Augmentation of the company's expert capability,
- Preserved expertise and
- To provide another programming tool, for the tasks which are difficult to do using standard programming languages (Knowledge Based Systems).

From the considerable number of projects currently underway, the problems associated with designing and building expert systems are now showing up, and are discussed in greater detail in Olson and Rueter (1987).

The problems fall into three categories:

- Knowledge Acquisition,
- Knowledge Representation and
- Presentation of the Knowledge.

- Knowledge Acquisition means extracting the knowledge from human sources or from records of past experiences. Techniques such as brain storming, group discussions and refined interview techniques improve the ability of the Knowledge Engineer to obtain the necessary knowledge for the expert system. Algorithms, have also been developed to extract knowledge from records by building decision trees from the raw data. However problems arise in building the best tree, especially when dealing with incomplete or erroneous data. (Discussions on these algorithms and induction in general can be found in Quinlan (1979; 1986a; 1986b; 1986c).)

- Knowledge Representation deals with the methods of representing the acquired knowledge in a machine. Factors to be considered include the suitability of the representation technique to the problem domain, maintainability and ease of writing and implementing. Current research looks at these different factors in, for example; rule-based systems, networks and frames, and tries to improve their ability to represent knowledge. The areas being covered include reasoning techniques and in particular uncertain reasoning. Work has also been done to improve the ability of the different techniques to represent knowledge.

- The conclusion of an enquiry session has to be presented to the user and the most common method is to use templates. Research is continuing on using natural language, so as to present the advice in the most natural way possible.

In this thesis, modules and partial rules are proposed as a means of improving the way that knowledge is represented in a rule-based expert system and an expert system, Aristotle, has been implemented to experiment with these ideas. Chapter 2 will look at some of the weaknesses in current programs and pin-point as precisely as possible what they are. Chapter 3 introduces modules and partial rules and demonstrates how

they fit into the expert system framework. Chapter 4 gives an in-depth introduction to Aristotle. Chapter 5 justifies modules and partial rules by looking at two large examples, and Chapter 6 looks at the future of Aristotle and modular expert systems.

CHAPTER 2 RULE BASED EXPERT SYSTEMS AND THEIR PROBLEMS

This chapter will look at the definitions found in Lansdown (1983) and Goodall (1985), which are representative of the many definitions published to date. Lansdown looks at the expert system from a behavioural point of view, and states how an expert system should look to the user. Goodall considers the functionality of expert systems and describes how they should work. Each definition will be considered in turn and the difficulties that arise will be described, followed by a discussion of those difficulties, which are based on the rule being the basic structure for representing knowledge.

Initially, Lansdown's definition will be discussed, point by point.

1 "Expert Systems know a great deal about a limited, but useful area of interest - such knowledge being acquired possibly from experience but, more likely, from human tutors."

If an expert system knows a great deal about a limited, but useful area of interest, then this information (usually a lot) must have been gained from somewhere. Lansdown names two sources which are:

- experience,
- expert human tutor

with the expert tutor being the most likely. This naturally raises the question - How hard is it to extract expert knowledge of a domain from a human tutor? Olson and Rueter (1987) discuss this and describe several techniques for extracting expertise from experts. In a comment on developing new expert systems, they say that the "...bottleneck in the development of expert systems is in extracting the knowledge from the expert..." (Olson, Rueter 1987). Obviously, this is one of the problems which face the designer of any expert system. It occurs because of the way that the expert stores his knowledge and uses it. To sum up Olson and Rueter, the knowledge is stored as extensive patterns in the mind of the expert, which are pattern matched with the current situation. For the expert, the set of patterns is richer than for the novice, and he is able to filter out bad matches far more quickly than the novice. (Also see Chase

and Simon (1986), deGroot (1965) and Reitman (1976) for further discussion).

The knowledge, which the expert has, is of many forms. In his day-to-day operations the expert would normally work on intuition or simple pattern matching, but when faced with a difficult problem has the full resources of his knowledge to fall back on. This knowledge can be divided into the following types:

- concepts,
- principles,
- algorithms,
- case histories and analogous reasoning,
- heuristic rules,
- known facts,
- expectations and
- intuition.

Devising a computer program that can handle all these forms of knowledge, and so imitate at least superficially the human expert's mind, is a difficult, and some would argue, impossible task.

The Rule-Based Expert System's approach is to code as many types of knowledge as possible into rules. Facts and expectations are simple to encode as their nature is simple. Heuristics are also suitable for encoding because they usually can be converted to rule form. Some of the others are impossible today, however. Intuition is an example; we do not know what it is or how it works, and so coding it is impossible. Using case histories requires analogical reasoning, a field in which very little is known, and learning algorithms, like ID3, have many limitations. The result is that it is not possible to encode all the types of knowledge, which an expert uses.

There are problems in extracting knowledge from a human expert, even for knowledge which can be encoded, because he is asked to express his knowledge in a form which is easily translated into rules. Problems which occur as a result of this translation can be:

- The knowledge which the expert gives may be out of date. (Alvey, Myers et al 1984). This is a common problem and is not simply a matter of selecting the right expert, although of course, this is important. It may be the case that the expert is, for some reason, deliberately holding back information, or he may not mention the latest development, because it is not fully understood by that professional community. For example, a new drug which seems to work, but no one knows why.
- The expert may alter his opinion. (Alvey, Myers et al 1984). Changes of opinion mid-way through the acquisition process may affect other parts of the knowledge. The expert may not realise the significance and the knowledge engineer would not normally have the understanding to notice the difficulty.
- The knowledge engineer may misinterpret the information given by the expert and code it up incorrectly. This is particularly difficult if it is an exceptional case, which may not come up very often during testing.
- The expert may oversimplify or over-generalise the knowledge when trying to explain the rules to a domain-illiterate knowledge engineer.
- The expert and knowledge engineer may inadvertently introduce a *supportive condition error*. The supportive condition is a condition, which is being used to support another condition, but which by itself has no significance. An example of this is the condition *blood-shot eyes*. When it is connected with *headaches and stomach trouble, blood-shot eyes* carries a heavier weight than it would by itself, while *headaches and stomach trouble* by themselves are always significant. The problem occurs because *blood-shot eyes* has a markedly different weighting in the context of *stomach trouble and headaches* than it has, if it occurred by itself. The knowledge engineer has to know this and treat the fact accordingly. The expert may not realise or not mention the nature of the condition, *blood-shot eyes*.
- The knowledge engineer may also introduce errors because of his naivity. He may consolidate several rules, which in most cases behave correctly, but not always. (Alvey, Myers et al 1984). This is called a *blunderbuss rule*.

In general these problems will be ironed out during the testing phase of the expert system's development. In summary, "the production rule format does not confer

immunity against errors..." (Alvey, Myers et al 1984) of the many kinds described above, and other tools and structures are needed to reduce the modify-test cycle of an expert system's development.

2 "They give their advice conversationally in the manner of a consultant, and can understand and respond to simple questions posed in plain (though perhaps specialised) language."

Building an expert system with these capabilities means that the knowledge must be represented in a manner which is suitable. This may not be the same as the one which is most efficient for representing the body of knowledge.

3 "Their knowledge is embodied not in the form of conventional programs but frequently by means of separate modules containing sets of rules with corresponding actions. This feature makes for an easier correction of deficiencies or errors in their knowledge-bases as well as in the acquisition of new knowledge. Strictly the implication of this is, that the knowledge (facts and inference) rules exist independently of the program. This makes it possible (theoretically, at least) to use the same program with a variety of knowledge-bases."

In this aspect of Lansdown's definition there is a way of reducing some of the difficulties of expert system development. Later on, this thesis will study how modularisation can achieve this.

4 "Because the areas of interest which they deal with are frequently ones where uncertainity prevails, expert systems often give their advice in probabilistic rather than absolute terms."

In many cases the expert is dealing with knowledge which is not fully understood. He often does not have a categorical rule which he can apply (Alvey, Myers et al 1984), but instead uses rules-of-thumb (heuristics). There may be doubts about the knowledge itself or about its application. In this situation it is impossible for the expert system to give absolute advice, and some form of uncertain reasoning must be used. The expert is then put in a difficult position; he must quantify, in some way, the uncertainities in his knowledge.

The expert would normally use such terms as *usually, sometimes, rarely, unusually* to express, in his own mind, his confidence in his knowledge. Experience enables him to judge what these different phrases mean in different contexts. Having to express them as a number will almost inevitably introduce behaviour, which is difficult to control. If an expert system does implement uncertain knowledge then it is said to be capable of performing *uncertain reasoning* (Keen, Williams 1984). As well as having uncertain knowledge, an expert system may also be faced with uncertain information. The user may not exactly know the data which the expert system requires.

There are various techniques for performing uncertain reasoning. They include Certainity Factors, Probabilities and Fuzzy Logic. Extensive research has gone into this matter but each solution has its problems. For example; probabilities, "...as a way of overcoming the problems of unknown data, are too complicated for the naive user." (Merry 1985). Fuzzy Logic is seen by many as being the best so far, though it is still is not quite right.

Linked with uncertain knowledge and reasoning is graceful degradation. The expert's knowledge degrades gracefully, not dramatically. He is usually capable of reaching some conclusions on matters which are outside his immediate domain of expertise. Frost (1987) states that, while experts use different strategies for different types of problems, there is graceful degradation for difficult atypical problems. As an example of what Frost is talking about; a general medical practitioner will refer a patient to a medical specialist if he is unable to diagnosis a problem, and as his knowledge degrades gracefully, he knows which specialist is most appropriate. At the moment, expert systems cannot do this very well.

Related to the problem of uncertain information described above, there is another problem. The user may be asked a question for which there is no sensible response except to say that the question is irrelevant. This can occur in the most well thought out system, because it is very unlikely that the expert will foresee every problem.

5 "The questions posed by expert systems are limited to ones which are relevant to a particular line of reasoning. Thus, if at any time the expert system decides that it has sufficient information to arrive at a conclusion, it does not continue to ask questions."

Part 5 of Lansdown's definition means that the expert system should query the user as little as possible. It requires the expert system to use every piece of information available to it in reaching a conclusion.

6 "Above all, expert systems can explain and justify their reasoning in such a way that experts can accept their credibility and non-experts can learn from them."

This means that the expert system has to keep track of its reasoning trail and explain it to the user, when asked. Of course it is not sufficient to just print out the rules used, not even in template form, some form of natural language is necessary. Furthermore the expert system needs to be able to answer users questions on why, as well as how.

Goodall (1985) provides a slightly different definition:

1 "An expert system is a computer system that performs functions similar to those normally performed by a human expert. "

2 "An expert system is a computer system that uses a representation of human expertise in a specialist domain in order to perform functions similar to those normally performed by a human expert in that domain."

3 "An expert system is a computer system that operates by applying an inference mechanism to a body of specialist expertise represented in the form of 'knowledge'."

Goodall's first point compliments Lansdown's second point. Goodall says that the expert system should perform the same functions as a human expert, while Lansdown only requires that the external behaviour be humanlike. This aspect of Goodall's definition can be taken at many levels. A strict interpretation would mean that the expert system has to imitate the human brain, whilst another interpretation would mean

that it has to be functionally equivalent to the human brain. Lansdown takes this view. At the moment we are not able to imitate the human mind, but we are capable of simulating a small part of its external behaviour.

Rules are potentially capable of simulating most of the heuristic knowledge which the human expert knows. The problem is that they do not always do so very naturally.

Goodall's second and third points say that the knowledge is represented explicitly in the program. There is a separate inference engine which uses the knowledge as data. This is different to what happens with conventional programming. In normal programs; knowledge of the domain is in the program itself and there is no separation between knowledge and the code.

Some of the problems arising from using rules are:

- Complexity and
- Consistency.

Some of the factors which contribute to the complexity of an expert system are:

- the large amounts of information involved,
- the inter-relationships between the information,
- the generally non-systematic nature of the information involved and
- the uncertainities in the knowledge.

The engineer has to process all the knowledge which the expert has, and produce rules to represent it. As described previously, extracting the knowledge is an error-prone operation, and as the knowledge grows, so does the number of errors. These errors will be harder to find and correct as the knowledge base becomes more complex.

Alvey and Myers et al (1984) have considered the problems of complexity and consistency, with respect to rules, and they say:

> "An expert system is not just simply a collection of individual rules. Each rule relates to some concept in the domain and must be consistent with, and complimentary to, the other rules relating to the same concept. Rules should

be created and revised as a unit and there should be no gaps or unwarranted duplications in the logic of the unit. The notion that individual rules can be added to an expert system without affecting the performance of the others is far too simplistic in practice." (Alvey, Myers et al 1984)

Pang and MacFarlane (1987) state that it becomes "...difficult to see the consequences of adding a new rule to the system. It may lead to an undersirable interaction and result in the knowledge base containing contradictions and circular results." For example, suppose the following 3 rules are in the knowledge base:

if A then B
if C then A
if A then C

They need not be close together, they may be spread out over the entire knowledge base, and this could mean over 200 rules; and so locating all instances of them becomes difficult.

All these problems are due to the complexity of the knowledge base and they may result in inconsistencies. Consequently there is a significant effort required in the testing stages of any expert system, as well as in its maintenance. The knowledge engineer is not capable of keeping all the threads in his mind at the one time, and is not usually competent to see all the ramifications which flow from the way he has designed the rules. The expert is also not usually competent to see the finer nuances of the way the knowledge engineer built the knowledge base.

Many authors in recent times claim that rules are inflexible and lack expressive power (Keen, Williams 1984), and are inadequate (Merry 1985). They are inflexible because they only allow two structures, the fact and the rule. A fact is in essence only a simple variable and hence can be given a value, while a rule is highly structured, and is of the form:

if condition then action

Such languages are very fine grained in the knowledge that they can represent, and do not provide the facilities for defining complex structures. Fikes and Kehler (1985), when talking about logic, said the very same thing and since the two are related (syntactically), it is also possible to apply their comments to productions as well. Pang and MacFarlane (1987) go further and state; that the expressive powers of rules are inadequate for representing concepts and relationships between objects at both a descriptive and control level, because the rules would become too complex and ungainly. They also go on to say that rule systems are slow, because of the extensive searching that is involved when accessing a large knowledge base.

In conclusion, the problems which arise in expert systems are due to the nature and structure of the knowledge which is being encoded. It is seen that acquiring knowledge is a non-trivial job, and requires extensive work by the knowledge engineer to correctly obtain it. Representing the knowledge as rules, brings out the weaknesses of rules. Finally, having to present the knowledge in a form most suitable to the application is a difficult task. Different tasks have different requirements, but there is only the rule as a means of encoding knowledge.

CHAPTER 3 SOLUTIONS

The Production System is just one of the many different implementation methods available for knowledge bases. It is not necessarily the best, and work by Reichgelt and van Harmlen (1985, 1986) has shown that a model should be chosen to suit the particular domain. A discussion on the various models and their applications is beyond the scope of this thesis, but further insight can be gained from the works of Reichgelt and van Harmlen (1985,1986), Barber (1984) and Stefik et al. (1982).

This chapter will look at improvements which can be made to the basic production system framework. These modifications should make the production system a more useful methodology, and reduce some of the weaknesses outlined in Chapter 2. It will start by looking in detail at some characteristics of a good expert system.

Based on the work of Gallanti et al. (1986) production systems should:

- enable a detailed description of the system in terms of components and connections to be encoded. For example, a system of pipes and valves would be encoded as a series of object pipes and object valves, and not just as a group of completely disorganised rules.
- enable a component to be described in terms of internal components, values of these components and parameters, which describe that part of the outside world which relates to that component. For example; the internal components of a pipe and valve system would be the pipes and valves, and the values of these components might be the flow through the pipes, the valve openings etc. The values from the outside world might include the flow coming into the component as a whole and the flow leaving. This is confirmed by Koukoulis(1985) in his paper on developing a frame-based fault diagnosis expert system.
- enable a non-author to maintain the the expert system. Many factors have a bearing on the ease of maintaining an expert system. They revolve around the related ideas of readability, simplicity and controlling interactions. Controlling interactions between components helps maintenance of the system by reducing the area, which the knowledge engineer needs to focus on, to locate a problem. For example; in a large and complex system of pipes and valves, if the knowledge engineer

can concentrate on just one particular group of components, then this will help him in his maintenance work. Simplicity depends to some extent, on how suitable the formalism used to represent the knowledge, is for the knowledge domain.

- degrade gracefully when data is missing, or when situations arise that were not previously foreseen and for which the expert system does not have any knowledge. For example; a flow from one of the pipes may be missing, for some reason, but a good expert system should be able to reach some conclusion, no matter how tentative. On the other hand, it might be that there is a leak which was not foreseen (deemed impossible), for which a good expert system should still provide some sensible answer.

Some of these goals are partly achievable with our current programming technology. This thesis will start by looking at the modular approach and show that modules are an effective weapon against complexity.

Modular Approach

The characteristics outlined above are symptomatic of and suited to a top-down approach to constructing a knowledge base. Furthermore, experience with software engineering would suggest that modular construction is an aid to fixing and reducing errors.

Traditionally, expert systems concentrate on a small aspect of the overall situation. For example; a fault diagnosis expert system only diagnoses faults and does not perform process management. By restricting expert systems in this way, there is a better chance that a useful system can be produced, because the knowledge base and control strategy can be designed to suit the particular domain.

By using modules it is then possible to describe the whole problem quite succinctly at the global level. The system can be described in terms of components and connections between the components, for example; in fault diagnosis a system may be described as a series of components, each of which have their own faults, and connections between components, which themselves have faults.

Each component is a mini expert system and can be described at that level by modules. It can consist of sub-components, connections and information passed in from the outside world, so if there are two identical components, then there is no need to have a duplicate set of productions in the knowledge base. This aids in maintaining the knowledge, by simplifying it and the interactions which can occur between the various parts.

Modules can also be used to help the expert system to degrade gracefully. They provide a handy unit, above that of a rule, to reason about. For example; a heater system may be one module of many in a chemical plant, which consists of a heater element, electronics to control temperature and solution to be heated. If it is not possible to determine exactly where in the heater system a fault has occurred, it may be possible to reason that there is a fault in the heater system, and hence guide an engineer to where the fault may be.

When a human expert is trying to find a fault, or solve a problem, he starts by looking at one part, usually that part which initially caused concern. If he is unable to find the fault, he then looks at neighbouring components and perhaps sub-components of that component. In examining the other sections he usually looks first at those, which are linked to the initial suspect component. It is very rare for an expert to look at the system randomly or the whole system from top to bottom. An expert system should reflect this attitude in the control strategy and the knowledge base design.

Parameterised modules seem natural for any expert system, which is dealing with a knowledge domain that consists of components. Parameterised modules mean that it is no longer necessary to duplicate rules, which are identical in structure but different in specific values, for example, *heater1_flowin* and *heater2_flowin* for two identical heaters. Inserting extra components is then very much easier for modular, as opposed to linear, knowledge bases, and would only require the knowledge engineer to instantiate another module and provide values for its parameters.

Some systems have gone part of the way towards having modular knowledge bases. Nexpert (Neuron Data Corp, 1976) is an example, as it has an action part which can read a knowledge base from disk into memory. Once in memory, its rules become part of the internal rule soup and it gives no significance to the fact that it was a separate knowledge base. Furthermore, if Nexpert needs to use any rules in that knowledge base, the knowledge engineer must ensure that it has been read in previously. This is not modularity in any true sense, it is separability. The knowledge bases are separate, they reside in different files, but when in memory, they become as one. It is analogous to include files in software engineering.

Personal Consultant Plus (PC+) (Texas Instruments, 1987) does claim a form of modularity based on frames, which are instantiated when required. What it does not provide, however, is any process for parameterising the frame. Information is passed around by use of global variables, and via a hierarchical structure for facts (parameters in PC+ terminology). This system, and many like it, provide varying degrees of modularity. Each recognises that one large knowledge base is unmanageable and should be broken up, but does not follow through with full modularity. In Aristotle, the principles of modularity are applied to produce a fully modular expert system.

Aristotle

Aristotle is an expert system shell, whose knowledge base is designed to operate on modular principles. The knowledge base and inference strategy are designed to satisfy the requirements placed on them by modularity. In the process modules provide benefits, which a knowledge engineer can use to make his expert system an easier system to build and maintain. Furthermore, modules provide a useful facility to ensure that an expert system's knowledge will degrade gracefully.

In Aristotle, the module is the second most important knowledge construct behind the rule. Modules provide the capacity to divide the knowledge up into related blocks and limit interactions between rules, when they are not desired. They also mean that careful thought must be given, by the knowledge engineer, to decide what data needs to be

transferred between modules. As Aristotle was designed for domains where the structure is static, then it is sensible to give Aristotle *Statically Instantiated Modules*. Information transfer can now be made quite sophisticated.

The design of Aristotle is based on the principles used for compiled languages. An expert system, is after all, only a specialised form of programming language. Like traditional programming languages, a good structure is essential if the expert system shell is to be easy to design, debug and maintain. Aristotle is a language which demands a disciplined approach to knowledge engineering.

Aristotle, as can be seen in figure 3.1, has a very definite structure. The knowledge base is a module which has several parts. Firstly, there are declarations; every fact and module must be declared and instantiated before it is used. Secondly, there are the rules, the first of which, is always the goal rule and it is followed by the normal rules.

```
Module Main:
<Partial Rule>
<Declarations>
        <Fact Declarations>
        <Module Declarations and Instantiations>
<Goal Rule>
<Rules>
endmod.
```

Figure 3.1 The Structure of an Expert System using Aristotle.

There are of three types of rules: goal, normal and partial. The goal and normal rules follow the declarations, the partial rule does not. The partial rule for each module follows immediately after the head of the module, this was a deliberate decision to

emphasise the different nature of the knowledge being represented. Combining the different types of knowledge into one single rule base, while feasible, does not highlight their differences.

Modules

Modules in Aristotle consist of two parts, a module declaration part and a module activation (instantiation) part. The module declaration has four sections. Firstly there is a heading which consists of the module's name, followed by a list of the parameters.

```
module pipe ( param flow_in, flow_out, flow_meter )
```

Then follows a declaration section where a partial rule is defined (see page 24). The third section is for declarations of facts, modules and module instantiations. The final section contains the rules, the first of which is always the goal rule.

When a module is part of the condition of a rule,

```
if pipe_1 then fault = T
```

then the expert system attempts to satisfy a module by attempting to satisfy the condition part of the goal rule of the module. The result of the condition part of the goal rule is the result of the module. It is guaranteed that whenever a module is applied then the goal rule is also attempted. There is no similar guarantee for any of the normal rules, and the significance of this will be seen in our discussion of partial rules.

The next aspect to be considered is the instantiation of modules. This is done with the *use* statement.

```
pipe_1 use pipe (argument var flow_in = ft34;
                         var flow_out = ft36;
                         var flow_meter = ft35);
```

where *pipe* is a module for pipes and is already declared. For a knowledge base to be truly modular, it should not communicate with the outside world except by using specified parameters. Using the pipe module above, only *flow_in, flow_meter* and *flow_out* are accessible from the outside. The interface is cleaner and easier to

maintain. A *var* parameter is always a simple fact, it cannot be an expression. It means that the expert system should not try to evaluate that parameter at that time, but should go straight to the *original* declared fact, and evaluate it. This is in contrast to *val* parameters, where the expert system will try and evaluate the fact immediately. This parameter may be a parameter itself and may mean further chasing of parameters down trails. This difference is important, because it is often desirable for the expert system to go back to the *original* fact and evaluate it, which may mean looking at a sibling module to the current module.

Consider an example (figure 3.2) of two pipes connected together with each pipe having an input and output flow and an internal flow meter. The knowledge for these pipes is coded in the module *Pipe*. In figure 3.2 there are the instantiation clauses for *pipe_1* and *pipe_2*. The output of *pipe_1* is connected to the input of *pipe_2*, because the output of *pipe_1* is *ft37*, which is the flow meter in *pipe_2*. Similarly, the input of *pipe_2* is *ft36*, which is the flow meter of *pipe_1*. In this way the two pipes are linked together.

To add a T-junction to this pipe system, as in figure 3.3, requires several modifications to the knowledge base. A T-junction module is written, which then is instantiated so that it fits between *pipe_1* and *pipe_2*. The T-junction module of figure 3.3 has four parameters, *flow_in*, *flow_out*, *t_flow_out* and *split_ratio*. A T-junction does not have any flow meters, but the input and two output flows are measured. It divides the flow between the outputs according to the ratio *split_ratio*, so that *split_ratio* percent is diverted along the trunk of the T.

The instantiations are different to reflect the new design. The output of *pipe_1* is the sum of the flow meter from pipe_1 and the output flow from the T-junction, that is;

```
pipe_1 use pipe (argument var flow_in = ft35;
                         var flow_out = ft37;
                         var Flow_meter = ft36)

pipe_2 use pipe (argument var flow_in = ft36;
                         var flow_out = ft38;
                         var Flow_meter = ft37)
```

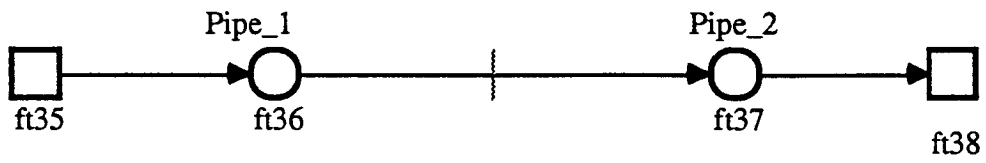


Figure 3.2 An Example Knowledge Base of two pipes connected together

(ft39 + ft37),

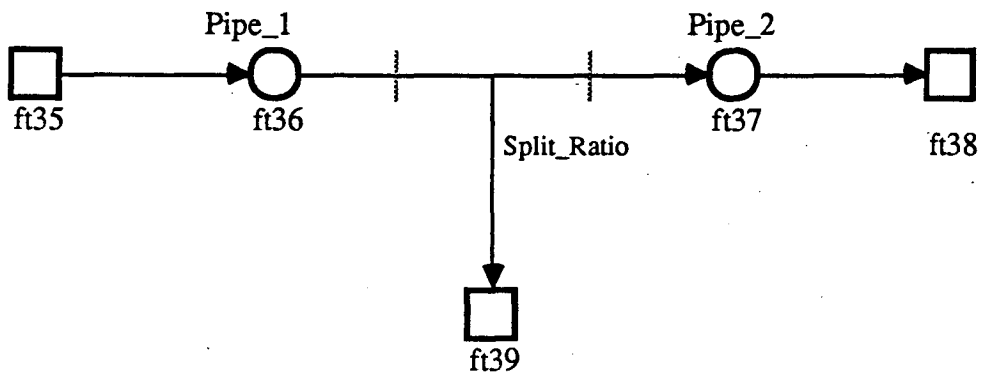
while the input to *pipe_2* is now the flow through the flow meter in *pipe_1*, less that which is being diverted,

(ft36 - ft39).

The flow out from the T-junction is the same as the flow into *pipe_2*, and the flow out from *pipe_1* is the same as the flow in to the T-junction. With this new knowledge base none of the rules were altered; only module instantiations were changed.

By parameterising the splitting ratio of the T-junction it is possible to use this particular module for different T-junctions, which can have different splitting ratios, and in fact, the splitting ratio need not be fixed. In a situation where the output of *pipe_2*

```
module t_junction (param flow_in,
                        flow_out,
                        t_flow_out,
                        split_ratio)
```




```
pipe_1 use pipe (argument var flow_in = ft35;
                         var flow_out = ft37;
                         var Flow_meter = ft36)

t_junc use t_junction (argument var flow_in = ft36;
                         val flow_out = ft36 - ft39
                         var t_flow_out = ft39;
                         split_ratio = 50)

pipe_2 use pipe (argument var flow_in = ft36;
                         var flow_out = ft38;
                         var Flow_meter = ft37)
```

Figure 3.3 Adding a T-Junction to the Knowledge Base in Figure 3.2

(figure 3.3) is diverted into one of two additional, unequal diameter pipes, (not shown in figure), then because of the different resistance in each pipe, the splitting ratio of the T-junction will vary. This feature can be incorporated into the knowledge base without too much difficulty by making the *split_ratio* variable dependent on where the

output goes. An example is shown in figure 3.4.

```
if diversion =  t then actual_split_ratio = 50
if diversion =  f then actual_split_ratio = 66

t_junc use t_junction (argument var flow_in = ft36;
                       val flow_out = ft36 - ft39
                       var t_flow_out = ft39;
                       split_ratio =
                                actual_split_ratio)
```

Figure 3.4 Alterations needed to figure 3.3 to allow varying splitting ratios.

These examples demonstrate how it is possible to build an expert system using modules as components and connections, thus satisfying the second criteria for a good expert system.

Modularity does not guarantee that the knowledge engineer can add rules with abandonment to one module without affecting other modules. If he changes the functionality of a module (changes its parameters) or the internal workings of the component, as reflected in the rules, then he is in effect creating a different component and this may mean other modules need to be changed. For example; replacing a valve which fails closed, by one which fails open, has a fundamental effect on all components which rely on the operation of the original valve, and so in effect, the component is no longer the same. In this type of situation modularity means that the changes will not be as difficult to make as they might have been with a non-modular expert system.

Graceful Degradation

A human expert's knowledge degrades gracefully. His knowledge extends beyond his limited domain of expertise into related disciplines. Naturally enough, he is not as familiar with these related disciplines as he is with his own, and so his advice in these areas is not as definitive as in his own. It is important that an expert system should have some of these qualities so that it will be useful in real world situations. Since expert systems are designed to be expert only in small and well defined domains, their spread of knowledge need not be as diverse as would be expected from a human expert. Despite this, extra knowledge is still needed to cover related subjects.

There are many ways that extra knowledge can be coded into an expert system. The principle used in the Naive Physics Manifesto (Hayes 1979) is to incorporate common sense knowledge, and then reason with it. Such common sense medical/engineering/metallurgical knowledge can be put into expert systems, but the question arises, where and how it should be used. Experience with the Naive Physics Manifesto shows that there is a lot of background knowledge needed, and so problems of speed and size arise, particularly for expert systems which are already big. Another option is to *smear* the knowledge to cover gaps in the knowledge and to extend it beyond its well defined boundaries, for example; probabilistic, fuzzy and certainity factor reasoning. These techniques have their advantages and disadvantages, but a complete discussion of them is beyond the scope of this thesis.

Probabilities have the advantage that they are based on sound mathematical principles and are well understood mathematically. On the negative side, the layman does not always understand them and may have difficulties in thinking in probabilistic terms. Certainity factors provide a simpler system for the layman to use, but they are not founded on any mathematical principles. Fuzzy Logic has a solid foundation, and is easier for the layman to use. As a logic though, it is not fully understood.

Aristotle has taken the Naive Physics approach to solving the problem of knowledge degradation. There is background or common sense knowledge in the knowledge base as well as the normal expertise. This extra knowledge is not complete, and is not as

broad as envisaged with the Naive Physics Manifesto. By following this approach, the other techniques in use today are not excluded, in particular, fuzzy reasoning may be used.

The most natural way of representing extra knowledge in a production system is to use productions rules, though the control strategy and general principles of use will vary. Normally the goal rule of a module is attempted, and if that rule is satisfied then the module is satisfied. If the goal rule is not satisfied then the module is not satisfied. As mentioned above, there is no guarantee that any of the rules will be used in the reasoning process, except the goal rule, which must be used exactly once. As a result, the *partial rule* was introduced to ensure that the *partial* knowledge (rules) will be used by the expert system. The *partial rule* is tried exactly once, and thus supplies the links necessary to use the extra knowledge.

Partial Rule

Before discussing partial rules, it might useful at this point to consider the purpose of expert systems as currently conceived. The purpose of an expert system is to provide the user with expert advice in a similar manner to a human expert. An example of this is a doctor who refers a patient to a specialist; or a specialist who refers a patient to another specialist. Each of these people know enough about disciplines, which are related to their own, to recommend another expert. If an expert system is to be accepted by the professional world it must have similar characteristics.

To date, very little has been done to give an expert system a broader knowledge base without overwhelming it with undesirable detail. If this is to be done, the question arises where to draw the line when putting in extra knowledge. Taking it to the extreme, means providing the expert system with common sense knowledge along the lines of a Naive Medical (Metallurgical, Engineering etc) Manifesto. To date, no true Naive Physics Manifesto system has been built, and there is no reason to assume that it will in the immediate future.

In an attempt to reach a balance between providing no extra knowledge and providing every conceivable related piece of knowledge to the expert system, Aristotle has introduced the *partial rule* and *partial conclusion*

A *partial conclusion* is a conclusion, which is of a more general nature than a normal conclusion, and which of itself, would not be sufficient to satisfy the competent user. For example; an Orthopedic Surgeon whose diagnosis is - *The patient has a broken leg,* would not inspire trust in himself from the General Practitioner. A more specific diagnosis would be expected. What the partial conclusion does, is to provide the user with enough information to enable him to decide what to do next. Normally in an expert system there would not be any reason to use this knowledge for inference purposes, *but* there is no reason why it cannot be so used.

The *partial rule* is a special rule which is used to control the extra knowledge and is similar in nature to the goal rule. There is, at most, one partial rule in each module, and that rule can refer to any fact defined in that module. Furthermore, the control system can use any rule to satisfy any condition in a partial rule, the exception being, partial rules are never used when trying to get a value for a fact. This limitation is justified by the nature of the knowledge which partial rules are introduced to handle. When the partial rule is fired, any conclusions reached are available for use in other rules just like normal conclusions.

Partial rules being of a more *general* nature than normal rules should not be mixed up with the goal rule in a well designed knowledge base. The *specific* knowledge, ie the goal rule, is physically separate from the partial rule. The normal rules are accessible to both the goal and partial rule, and even though a partial rule can use the normal knowledge available, the knowledge engineer is not encouraged to let a partial rule use normal rules for reasoning. The occasion may arise when the partial knowledge for a module is too complex for a single rule, in which case extra rules may be written which live with the normal rules, but which, by use of the goal-directness of Aristotle, are only used by the partial rule and never by the goal rule. This is only a restriction on the knowledge engineer and not on the control strategy, so if the knowledge

engineer wishes, he can allow a goal rule access to rules associated with a partial rule and conversely rules associated with a goal rule accessible to a partial rule. Appendix E is an example where normal and partial knowledge co-exist within the same rule base. It is up to the programmer to exercise discipline and only mix the two in a controlled manner, where necessary. There are differences, from the reasoning point of view, between normal and partial rules. They lie in what is produced by an expert system at the end of a consultation and the very nature of the knowledge, which partial rules were introduced to handle. At the end of a consultation the user is informed of any final conclusions and any partial conclusions reached. The responsibility then falls on the user to make a decision based all the information provided to him.

Naturally, deciding what knowledge should go with partial rules and conclusions and what should go with normal rules and conclusions, is very much dependent on their applicability. It is up to the knowledge engineer to decide such matters, in consultation with the potential users and the domain expert. As a general rule though:

> If the knowledge is of a nature which will satisfy the user's query, then such knowledge is normal knowledge. If, on the other hand, the knowledge is not acceptable in itself, but is only a reference to other sources or is of too general a nature, then it should be considered as partial knowledge.

So, for example; deducing that there is a leak in Pipe 2 would be normal knowledge, but reaching the conclusion that there is an inconsistency between the input and output flows, without saying whether there is a leak or a faulty meter, would be partial knowledge.

The structuring of the knowledge base determines to a large extent the usefulness of partial knowledge. With the examples used in this thesis, the most natural way to break the domain up is to split it into components such as pipes, T-junctions etc. The partial rule for each module would reflect this structure and provide extra knowledge on each component. Particular examples will be discussed in the following chapter, but figure 3.5 below, will illustrate the type of knowledge which could be associated with a partial rule.

```
partial : if (flowin <> (flowmeter1 + tflowout)) cor
             (flowmeter1 <> flowin) cor
             (flowmeter2 <> flowout) cor
             (tflowout <> flowmeter1 / 2))
          then perr = t;


fact perr:
   type : logical;
   status : infer;
   expln : 'Problem with the line';
```

Figure 3.5 An example partial rule for the pipe system of Appendix E

This rule says that if the readings from various flow meters do not add up to the correct values, then there is a problem with this particular line. There might be a leak or a faulty flow meter or there might be a combination of both problems. Such a simple diagnosis might not be obvious when this pipe system is just one part of a larger more complex pipe system. So, even though it is not particularly helpful as far as pin-pointing the actual fault in the whole plant, it is useful in narrowing down the problem area to a particular pipe or group of pipes.

Logic and Reasoning

Classical logic has been shown to be too inflexible when attempting to model the real world. As was discussed in Chapter 2, it is quite often the case that the user or knowledge engineer is working with incomplete information. Earlier discussions and the work of (Gordon and Shortliffe 1985; Shafer 1976; Shortliffe and Buchanan 1975; Duda et al. 1976; Buchanan and Duda 1983; Gaines 1976.) point out that probabilities, certainity factors and fuzzy logic have some major drawbacks.

At the moment, work is continuing on fuzzy logic as a means of providing a sound foundation for uncertain reasoning.(Zadeh 1965, Zadeh, Fu et al. 1975.) Each system however uses the logical values, true and false, for their reasoning. For the real world this is too restrictive. The user of an expert system is often in a situation where true or false is not appropriate; not because the user is unsure of whether something is true or false, but because it may not be relevant to the current situation at all.

An example is a a doctor asking a blind patient if he gets headaches when he is reading. There is no sensible answer and *doubtful* does not make sense in this context. If the patient had sight, then doubtful may be a sensible answer in certain circumstances. With a patient who is blind, however, the question is nonsense, and an answer like *doubtful,* is misleading. It is therefore sensible to provide a third logical value, *irrelevant.*

It is not acceptable for a doctor to ask every patient who comes into his surgery if he is blind. Similarly, it is not acceptable for an expert system to do so either. One of the requirements of an expert system is, that it should behave in a manner which is as close as possible to that of the human expert. The human assumes the obvious until he has some evidence (or an intuitive feeling) that his assumptions are not correct. Most people, and therefore most patients, who visit a doctor will not be blind. The doctor does not doubt this assumption until a patient exhibits symptoms which are suggestive of a blind person.

Of course a human has five senses to help in this matter. The computer usually has only one way of sensing the world - that is the keyboard - which is not very expressive. If the expert system is to copy the human's public deductive processes, then it must make the most of whatever information it has available, especially with regard to basic assumptions. A response of *irrelevant* to the question about headaches, when reading, would suggest, that the assumption of sight or literacy may be in need of substantial modification. An answer of *true* or *false* to the question - "do you get headaches when reading", is sufficient to indicate to the doctor that the patient is not blind, though in fact this is not questioned, unless there is cause to do so. A response

of *do-not-know,* again suggests that the patient has sight, but is not very observant, or has a lot of headaches and cannot associate them with any particular activity. It does not suggest that the patient is blind.

Aristotle uses a five value logic. The values are:

- *true,*
- *false,*
- *do-not-know,*
- *unknown,*
- *irrelevant.*

True and *false* are interpreted as they normally would be in classical logic, while *irrelevant* is seen as being neither *true* or *false*. The clause or question just does not apply. *Do-not-know* is viewed as either *true* or *false,* but the question or statement is relevant. *Unknown,* on the other hand is more general, this says that the result may be either *true, false* or *irrelevant.* It is a more general form of *do-not-know,* and includes *irrelevant.*

This logic is based on the work of Kleene (1938, 1952.) where he outlined a three value logic using T F and I. The relationship between the five logical values can be seen in figure 3.6.

The truth tables for the 5 value logic can be seen in figure 3.7. This logic is monotonic and improves the flexibility of the logic system, something which is necessary for a real life system.

It now remains to show that the logic is sensible. The operators are defined so that they have, where practical, the same definitions as exist in classical logic. Therefore:

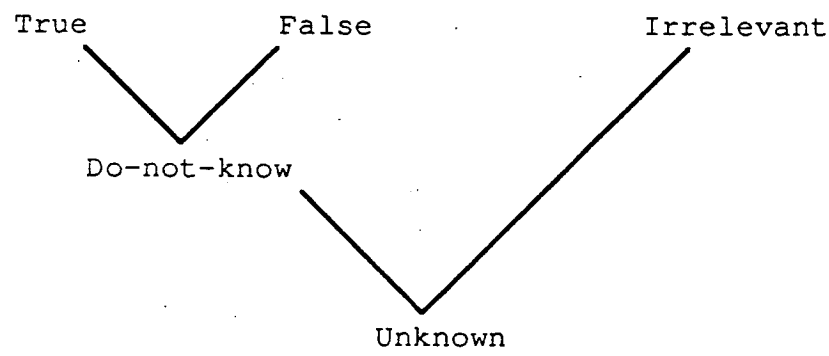


Figure 3.6 The relationships between the five logical values.

or	t	f	d	u	i
t	t	t	t	t	t
f	t	f	d	u	i
d	t	d	d	u	u
u	t	u	u	u	u
i	t	i	u	u	i

not	
t	f
f	t
d	d
u	u
i	i

and	t	f	d	u	i
t	t	f	d	u	i
f	f	f	f	f	f
d	d	f	d	u	u
u	u	f	u	u	u
i	i	f	u	u	i

Figure 3.7 Truth Tables for the five-value logic.

(1) A and B = (˜A) or (˜B)

(2) A or B = (˜A) and (˜B)

(3) A implies B = (˜A) or B

(4) A = ˜(˜A)

(5) A equivalent B = (A implies B) and (B implies A)

and the logic is defined using *or* and *not.*

NOT

Looking at the *not* truth table, *not true* is *false* and *not false* is *true. Not,* defined on *true* and *false,* behaves exactly like the *not* of classical logic.

Consider the negation of *do-not-know* and *unknown. Do-not-know* means that the statement is relevant, but it is not known whether it is *true* or *false;* therefore *Not Do-Not-Know* intuitively means that the response is known, but it can be either *true* or *false.* Since it is not known which one it is, then intuitively, this means that it is *do-not-know.* More formally, assume that the negation of *do-not-know* is *true,* then

do-not-know = ˜(˜do-not-know) (4)

do-not-know = ˜(true) since ˜do-not-know = true

do-not-know = false since ˜true = false

which is a contradiction. A similar argument holds for *false* and *irrelevant.*

Not Unknown using a similar argument, means that it is not known what the logical value is. It could be *true, false* or *irrelevant,* but since again, it is not known which one it is, then *not unknown* is *unknown.*

The final logical value to be considered is *irrelevant. Not irrelevant* can be one of five values, if it was *true* then by (4)

irrelevant = ~ (~ irrelevant)

irrelevant = ~ (true) since ~ irrelevant = true by definition

irrelevant = false, since ~ true = false,

which is a contradiction. A similar argument holds for *false, do-not-know* and *unknown*. Therefore *not irrelevant* must be *irrelevant.*

OR

With the *or* operator, *true or* any other logical value is *true,* in particular, *true or irrelevant* is *true*. Consider an example,

head_lights_ok or spot_lights_ok implies some_lights_work

if the head lights work, but there are no spot lights, then there are still some lights working, so *head_lights_ok or spot_lights_ok* should be *true*. So it is sensible to make *true or irrelevant, true. False* and any other logical value returns the other logical value, so *false or true* is *true,* and *false or irrelevant* is *irrelevant. False or do-not-know* means in effect, *false* or perhaps *true or false,* but it is not known which. Therefore the result could be *false or false* which is *false,* or it could be *false or true,* in which case, the result is *true*. Therefore *false or do-not-know* is *do-not-know*. A similar argument applies to *unknown,* so *false or unknown* is *unknown.*

A similar argument applies to produce the rest of the table in figure 3.7. By applying the rules above, you get the *and, imply* and *equivalent* tables, which are not reproduced here. This logic is sensible and applicable, as seen earlier; furthermore it is more closely aligned to the human expert's thought processes than classical logic, as may be seen in Chapter 5.

In conclusion, modules and modularity provide a technique for dividing up the knowledge base into sensible blocks, and controlling interactions between these blocks, while still maintaining some of the power of rules and production systems. The module is a handy object to store partial knowledge and ensures that the expert system's performance may degrade gracefully. Finally the multi-value logic provides a

logic, which is capable of representing more complex notions in a better fashion than classical logic, and which, itself, is soundly based.

CHAPTER 4 ARISTOTLE IN DEPTH

This chapter will discuss aspects of the syntax and semantics of the goal-driven expert system shell, Aristotle. It will look at modules, facts, rules and conclusions and will proceed from there to look at parameters and the scope of facts and modules and then partial rules. As there are many expert systems and programming languages around it will not dwell on the more mundane aspects of Aristotle, like evaluation order, but will concentrate on those features which make Aristotle unique. This chapter will finish with the BNF definition of Aristotle. The five-valued logic introduced in chapter 3 will be demonstrated in detail in the next chapter.

Aristotle, consists of two sections; a compiler written using *lex* and *yacc* to produce MU-Prolog[1] clauses, and an interpreter written in MU-Prolog. Both sections run under Unix[2] on a Vax 11/750[3] at the University of Tasmania.

Knowledge Base

A Knowledge Base in Aristotle is a module which consists of three distinct structures which together are used to represent knowledge. These are facts, rules and modules. Facts and rules are used to code the knowledge, while modules are used to group related facts and rules in a coherent fashion.

The different roles played by the different structures are worth further discussion. A *fact* is a piece of information. For example; the fact:

Flow = 35

could represent a flow of 35 cubic meters per hour along a pipe. A *rule* is a piece of knowledge which allows an expert system to deduce the status or value of a fact based on the facts, which already are known, or facts for which an expert system can get a value. The form of a rule is:

---

[1] Melbourne University
[2] Trademark AT&T
[3] Digital Electric Corporation

if <condition> then <conclusion>

For example:

if flow > 110 then flow_status = addmem error

There are three types of rules allowed in Aristotle. They are:

- normal rules,
- goal rules and
- partial rules.

Each has the same structure though, the purpose and use for each is different. Each type of rule will be discussed later in this chapter.

The final piece of the knowledge base to be considered is the *module* which contains all the knowledge relating to one particular item or group of items. For example; in an expert system which diagnoses faults in heater systems, there is one module which contains all the knowledge. This is the main module or *knowledge base*. A main module contains sub-modules which group sub-components together. For example; one module for each individual heater.

Modules

The knowledge base of Aristotle is a module called *main,* within which there are rules, declarations of facts, modules and instantiations of modules. A declaration of a module (generic module) can be seen in figure 4.1.

<Declarations> consist of the declarations of facts, generic modules and module instantiations in any order, so long as the object is declared before it is used. An exception to this is a partial rule, where conditions and conclusions are declared after they are used.

When a module is instantiated it is made *active*. When instantiating a module, values for each parameter are specified. The *Main* module is instantiated automatically when an enquiry session commences, so there is no need to instantiate it. Instantiations are

```
module <name> ( param <parameter list> ) :
partial : <rule>
interface
<declarations>
goal : <rule>
<rules>
endmod;
```

Figure 4.1 The declaration of a module in Aristotle.

of the form:

<inst_mod> use <generic_mod> (argument <argument_list>) : <expln>;

eg pipe_1 use pipe (argument var flowin = ft35;
var flowout = ft37;
var flowmeter = ft36) : 'Pipe 1';

The argument section associates the formal parameters, *flowin, flowout* and *flowmeter,* with the appropriate outside values *ft35, ft37* and *ft36*. This association remains for the lifetime of the expert system. An argument may also be an expression rather than just a simple fact. For example:

flowout = (ft35 + ft45);

would result in the following declaration:

pipe_1 use pipe (argument var flowin = ft35;
val flowout = (ft35 + ft45);
var flowmeter = ft36) : 'Pipe 1';

where instead of *var* we now put *val. Var* and *val* derive from software engineering. A *var* parameter, when altered, effects the outside world, but a *val* parameter does not. The same occurs in Aristotle. A *var* parameter can pass a value in or out, while a *val* parameter can only pass a value in. More importantly, the parameter trail does not

extend past a *val* parameter. Parameter trails will be discussed in greater detail later in this chapter.

In a rule, reference is always made to an instantiated module, never to a generic module.

```
    if pipe_1 then ...
never
    if pipe then ...
```

The module is used in a similar fashion to the function in third generation languages. A module returns a result which is the result of the goal rule, which in turn is the result of the condition of the goal rule. For example;

Using figure 4.2; in trying to find a value for *poss_fault_part,* and assuming that *pipe_used* equals 1, it is now necessary to attempt the module *faulty_pipe_1*. If that returns true, then we can add *pipe1* to the set of possible faulty parts *poss_fault_part*. To satisfy *faulty_pipe_1* it is necessary to try the goal rule of *faulty_pipe_1*. If *leak or blockage* returns true, then the goal rule is satisfied; which means that *faulty_pipe_1* also returns true, and hence rule 93 is satisfied, and *pipe1* is added to the set *poss_fault_part.*

This is different to most other expert systems which treat modules, frames and rule groups as procedures. Where reference is made to modules in the action (conclusion) part of a rule, those modules do not return any result. They are used to acquire knowledge, not to participate in the process itself.

A module is a piece of knowledge, admittedly complex and highly structured, but never-the-less, only a piece of knowledge. It can be summed up in one single fact, for example; the fact, *pipe_1_fact* can represent the module *pipe_1,* which determines if there is a fault in the first pipe. Now this module has a side effect of also saying what the fault is, via parameters. For example;

```
module main:

module generic_faulty_pipe (...

goal: if leak or blockage then ...
1   : if ...
  :
endmod;

faulty_pipe_1 use generic_faulty_pipe(...

goal: if ...
   :
93: if pipe_used = 1 and faulty_pipe_1
    then poss_faulty_part addmem pipe1:
   :
endmod;
```

Figure 4.2 An example of how modules are used as part of the reasoning process.

```
fact cause:
   type : member;
   value : [leak,blockage,faulty_sensor];

module pipe (param ...
                cause,
   :
31: if flowin > flowmid then cause = addmem leak
   :
endmod;

pipe1 use pipe (...
                var cause = cause;
   :
95: if pipe_1 then ...
97: if [leak] subset cause_pipe_1 then ...
```

The module *pipe_1* has determined that there is a fault, and that fault is a leak. This information is passed back by the parameter *cause,* which has the value *leak.* This is a side-effect. Of course, if there are no modules, then the same effect can still be achieved by following essentially the same reasoning path. Chapter 5 discusses this in more detail.

Side-effects are permitted in Aristotle. They are an important part of the control strategy, but they are controlled. Scoping, as we will see, means that unless a fact is a parameter, its scope is only the declaring module, therefore side-effects can only occur through parameters. Not to control them at all will mean that a single rule could effect the whole knowledge base, and for a large knowledge base this becomes quite a problem. The question then must be decided, how much side-effecting will be allowed? In a modular knowledge base, a suitable barrier is the module, which is what Aristotle has provided.

Scope

All module names in Aristotle must be unique. A module can only instantiate a child module, or the sibling of a parent or grand parent, whether it was declared previously or not. It is not possible to instantiate a module recursively, as this makes no sense when talking about a physical component. A module cannot instantiate the child of a sibling, or the child of a parent's sibling. These rules are similar to those of Modula-2 etc.

Looking at figure 4.3; the module *D* or *C* can instantiate *A* or *B, B* can instantiate *A* or *C,* but *A* or *B* cannot instantiate *D*. *C* can instantiate *D,* but *D* can not instantiate *C*. Furthermore, *D* cannot instantiate *E,* because that is the child of a parent's sibling.

A fact is only visible in the module where it was declared, though it can be passed in or out via a parameter.

Figure 4.4 is a valid knowledge base in which five facts are declared; each called *X,* but each is different. In figure 4.5; the fact *Y* declared in *A* is not accessible in *B*. Each

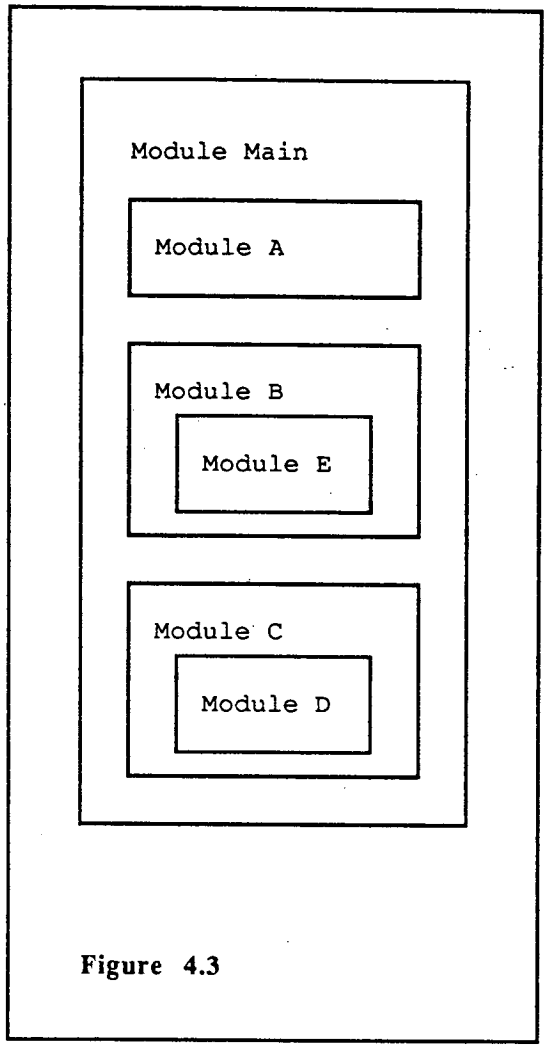


Figure 4.3

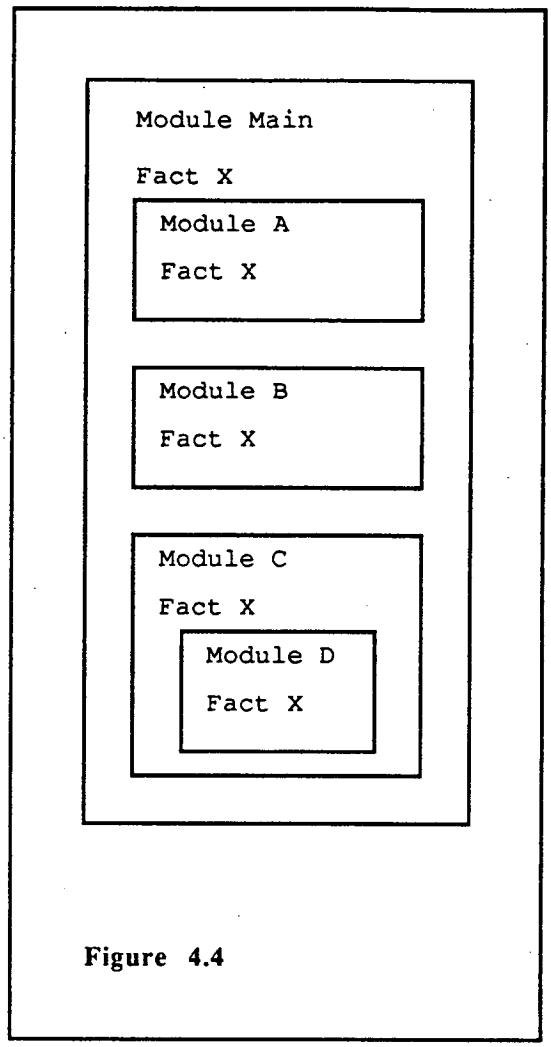


Figure 4.4

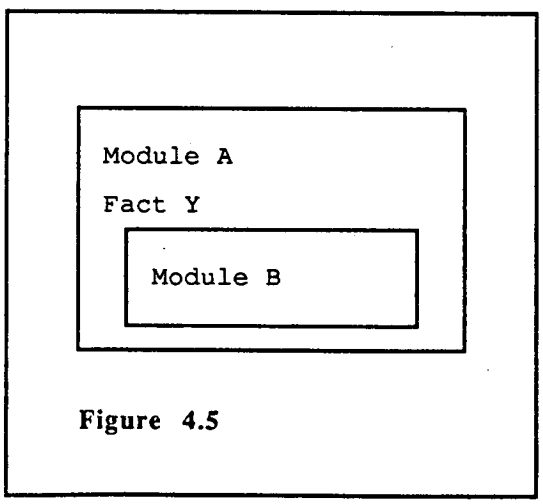


Figure 4.5

module is completely isolated from all other modules and the only way to pass information around, is by use of parameters. Facts and modules exist for the life of the expert system.

Parameters

Parameters provide the connections necessary to write modules, instantiate them, and use them several times. They provide the mechanism to pass knowledge between modules.

An argument is that value passed in from the outer module. The argument can be of two forms. The first, and conceptually most simple form, is when the argument is another fact.

flowin = ft35

This is a *var* parameter.

The second form is when the argument is an expression which needs to be evaluated. This is a *val* parameter

factor = 5
ratio = flowin / flowout
highflow = flowin > 500

The expression may be any legal expression, in particular it may be a single fact of the form

val flowin = ft35

The effect of this is to limit the areas where the control system searches for a rule. Since this is a *val* parameter, the control system will either get the value of *ft35*, if it is known, or accept that *flowin* is undefined.

The concepts of *parameter trail* and *search order* are very important in Aristotle. They provide Aristotle with the power necessary for it to modularise the knowledge base into separate components, while still allowing the components to interact with

each other.

In figure 4.6, the fact *X* is a formal parameter in module *C*. It shows that the actual parameter, or argument is *Y*, which is a parameter to the module *B*. The argument to *Y* is the fact *Z*, which is declared in the main module. The value of the fact *X*, then, is the same as the value of the fact *Z* in the main module, and the parameter trail is *X*, *Y*, *Z*.

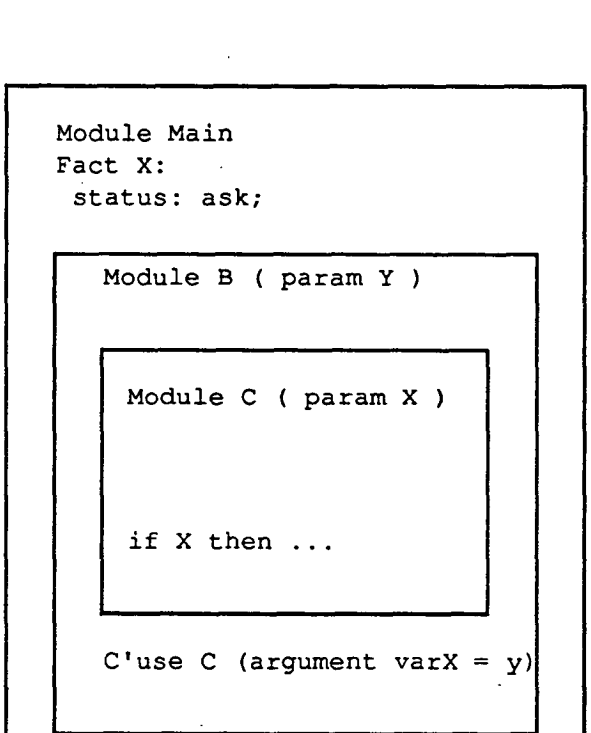


Figure 4.6

If a reference is made to *X*, then the goal-directed nature of Aristotle demands that an attempt be to find a value for it. If it is not known, then Aristotle must try else where. One of the two options described earlier is to infer its value, by using the rules

in the knowledge base, and the question then arises, where to find these rules. The two most obvious places are; firstly, the current module, and secondly, the module where the original fact was declared if it is a parameter; although these are not always enough. Consider the example of a system of pipes, where the output of one pipe is the input to a second. If the output of the first pipe is not known then it may be possible to obtain this information from the output of the second.

In figure 4.7 the input flow to the pipe *pipe2* is determined by adding the output flow and the flow meter in the middle of the pipe as seen by rule 22 of pipe2.

In module *pipe1*, the value of *pipe_out* can only be determined by getting the value of *pipe2_in*. If there are no rules in the main module which give *pipe2_in* a value, then all those modules, which have *pipe2_in* as a parameter are searched. In this case the module *pipe2* has *pipe2_in* as a parameter. The rules in this module are then searched to find if one them sets the value of *pipe2_in*. The control system finds that *pipe2_in* is an argument for *pipe_in* in *pipe2*, so it searches for all those rules in module *pipe2*, which have the fact *pipe_in* as a conclusion. In this case it would pick up rule 22 and the process then continues because it has to satisfy the condition of rule 22.

To find the value of a fact, the rules in the module where the fact was originally declared are considered, and any rules that may produce a value are tried. If no value, or an unsatisfactory value is found, then all sub-modules, where the fact is an argument, are tried in a depth-first fashion.

For *val* parameters, Aristotle does not follow the parameter trail. For example;

```
Module two_pipe_system ( param pipe_in, pipe_out, pipe_mid );
   :
fact pipe_in:
status: param;

fact pipe_out:
status: param;

fact pipe_mid:
status: param;
   :
goal: ...
   :
22: if <condition> then pipe_in = pipe_mid + pipe_out
   :
endmod;


pipe1 use one_pipe_type ( argument var pipe_in = ft35;
                                   var pipe_out = pipe2_in):

pipe2 use two_pipe_type ( argument var pipe_in = pipe2_in;
                                   var pipe_mid = ft37;
                                   var pipe_out = ft38):
```

Figure 4.7

pipe2a use two_pipe_type (argument val pipe_in = pipe2_in + ft40;
var pipe_mid = ft41;
var pipe_out = ft42)

pipe_in has a *val* parameter for an argument, whereas *pipe_mid* does not.

Aristotle would not try *pipe2a,* because *pipe_in* does not have a simple fact for an argument; but an expression. The control system finds the parameter *pipe_in* in *pipe2a,* has the argument

pipe_in = pipe2_in + ft40

and the control system would immediately try to get a value for *pipe2_in* and *ft40*. The parameter trail finishes.

Rules

A rule in Aristotle is of the form

rulenumber : if <condition> then <conclusion> : <explanation>

where a <condition> is an expression returning a logical value. If the result of <condition> is *t* (true), then <conclusion> is concluded. An expert system will try to evaluate the <condition>. If this is not possible, because part of the <condition> does not have a value, then an expert system will try to obtain a value for that particular fact. This is done in two ways, either by asking the user, or by trying to find a rule which will give the unknown fact a value. This fact becomes the new *goal*. The knowledge base is searched to find those rules, which give the goal a value. Normally several rules would be found, so an expert system tries the first. If this new rule's <condition> evaluates to *true,* then the <conclusion> is concluded, and the goal is satisfied. If on the other hand, this <condition> does not evaluate to *true,* then the next rule found is tried. This process continues until a value is obtained for the goal, or there are no rules left. If there are no rules left, then it is automatically concluded that the goal has a logical value *u* (unknown). Taking a motor car as an example:

1 : if running_hot is t then faulty_radiator = t

2 : if gauge_past_half then running_hot = t

3 : if steam_coming_from_under_the_bonnet then running_hot = t

If it is not known if the gauge is past half way, or if steam is coming from under the bonnet, then rules 2 and 3 will not be able to conclude that *running_hot* is *true*. In this case the expert system will automatically conclude that *running_hot* has the logical value *u* (unknown), and consequently, it will not conclude that *faulty_radiator* is *true*.

As was mentioned earlier, there are three types of rules. They are:

- normal,
- goal and
- partial.

There is exactly one goal rule, at most one partial rule and any number of normal rules in every module. The result of a module is the result of the goal rule, which is the result of the <condition> of the goal rule. This means that there is no real need to have a conclusion, when defining the goals of a module or knowledge base. For example many expert system shells only have goal conditions:

goal1 and goal2 and goal3 ...

By having a goal rule instead of only goal conditions, Aristotle is able to deduce new knowledge during all stages of the inference process. For linear knowledge bases this is obviously unnecessary, but since modules and hence goal rules occur many times during the inference process of a modular expert system, then it may be used by the knowledge engineer to compact the knowledge, while still keeping it readable.

A similar argument to the one used above can be used with partial rules. It is not necessary for the *partial rule* to have a <conclusion>, instead only partial conditions are necessary. In many cases however, the partial knowledge can be encoded into one single rule, or there may be only one partial conclusion.

Facts

A declaration of a fact consists of the name of the fact, its type, value range, status or origin, question string and explanation string.

```
fact <name> :
   type : <type>;
   value : <value>;
   status : <status>;
   question : <question string> ;
   expln : <explanation string>;

for example;

fact flow:
   type : number;
   value : 0..90;
   status : ask;
   question : 'What is the flow of solution through the pipe';
   expln : 'flow of solution through the pipe';
```

If the fact has not been given a particular value, then it is undefined and has no value. Initially, every fact is undefined. All facts, no matter what type, are defined if their value is *unknown* or *do-not-know,* including *number* and *member* type facts.

The types in Aristotle are :

- number,
- logical and
- member.

The type *number* consists of the integers and *d,u,i*. Number only contains integers because MU-Prolog does not implement floating point numbers. The following arithmetic operators +, -, /, *, =, <>, >, >=, <, <=, *bt, nbt* are in Aristotle. The brackets, (and) are used as normal to group expressions for reasons of clarity or precedence. The operators *bt* and *nbt* are *between* operators. They are defined to mean;

X bt Y and Z
(X > Y) and (X <= Z)

X nbt Y and Z
not (X bt Y and Z).

cand and *cor* are conditional operators. If the first operand of *cand* is false, then the right is not evaluated. If the first operand of *cor* is true, then the second operand is not evaluated. In all other cases both operands are evaluated. Expressions in Aristotle are of the form:

<a> <operator> <b>

or

<operator> <a>

<a> and <b> can both be expressions in themselves. If only one side returns *d,u* or *i,* then that is the result. If both sides return *d,u* or *i,* then the result is the result of the *equivalent* operator, which is defined to be:

(A implies B) and (B implies A)

A fact may also specify a value range, though it is not mandatory. A range is specified by

<integer>..<integer>
eg 1..100
-50..50

[<elt>,<elt>,..,<elt>]
eg [red,blue,green,orange]

The type *logical* has five values. They are *true (t), false (f), do-not-know (d), unknown (u)* and *irrelevant (i).* These are represented as single letter atoms in Prolog. The operators allowed on logical facts are: *is, and, cand, or, cor, not* and are defined by a series of Prolog clauses. (See Appendix L)

The only other operator defined for a logical expressions is the *is* operator. For example;

isleak is t
isadult is u

A fact may also be of type member. The operations allowed are a subset of the set operations. A set is represented as a list of atoms in Prolog. The atoms that are allowed may be defined by the *value* attribute, when the fact is declared.

eg. [blue, green, orange, red, white]

If no values have been specified in the values part of the fact's declaration, then any legal Prolog atom is allowed. As with *all* the other types in Aristotle, the logical values *d,u* and *i* are standard members of this type.

There are two operations allowed with facts of type member. They are:

- add an element
- test for subset

Elements can be added to the value of a fact only as part of a conclusion of a rule. For example;

if <condition> then leaf_colour = addmem red

red is now added to the set, *leaf_colour*. It does not remove the old value, as the other types do, so if before we tried the rule, the value of *leaf_colour* was *[blue,green],* afterwards it becomes *[blue,green,red].*

A test for subset is made by use of the *subset* operator. It is of the form:

skin_colour subset [pale, yellow, red, normal]
fluid_colour subset valid_colour

If either side of the *subset* operator has one of the values: *d,u* or *i,* then the result of that expression is always *d,u* or *i,* depending on which is logically the best result.

Aristotle demands that the elements of a set be fully specified. Sets of numbers are specified by sub-ranges. For example;

[1..25]

and sets of logical values by

[t,f,d,u,i]

Therefore,

```
fact skin_colour:
    type : member;
    value : [pale, yellow, red, normal]
    :

if [blue] subset skin_colour ...
```

is illegal Aristotle.

For all types of facts an explanation string is necessary. This string is used by the user interface whenever:

- the user is asked for the value of a fact,
- whenever the system is explaining its reasoning or
- when the system is producing answers at the end.

If the value of a fact is undefined, then a value can be found from two sources; the user or a rule. Its status attribute specifies its source. There are four status:

- ask,
- infer,
- both and
- param.

If the status is *ask,* then the question string is used to ask the user to enter a value, and no attempt is made to infer a value by using rules from the knowledge base. The status *infer* tells the system to search for a rule, whose conclusion may give the fact a

value, and not to ask the user. The status *both* tells the system to try and infer a value, and if it cannot, then and only then, it should ask the user. The final status *param* tells Aristotle, that this fact is a parameter and to get the value from the argument.

An expert system written in Aristotle should not, as one of its principles of design, query the user excessively. A similar principle also exists when designing an expert system, but this needs to be counter-balanced however, by the guiding principle common in software engineering, that it is better if the programmer defines the characteristics of a variable and a procedure so that typing errors can be found. When declaring a fact a knowledge engineer need not list all its properties. A fact, which is only to be inferred, need not have a question or prompt string. On the other hand it does require a knowledge engineer to specify the type of the fact, even though this can be inferred from context. The effect of this is to provide the knowledge engineer with a package which is able to detect a lot of the common typing errors, without being too demanding and wordy.

Conclusions in Aristotle

There are three types of conclusions in Aristotle. They are:

- ordinary conclusions,
- partial conclusions and
- final conclusions.

Each of the three types can occur in the knowledge base. Ordinary conclusions are the backbone of the reasoning process and are used to hold intermediate results which are not of importance to the user, because there is no facility to provide this information to the user.

Partial conclusions behave in almost the same way as normal conclusions with the added feature that at the end of the consultation this knowledge is presented to the user in an appropriate fashion. Final conclusions are almost identical to partial conclusions except the way in which the knowledge is presented to the user is different. Also the nature of the knowledge itself is different. A *final conclusion* is a conclusion which

achieves the purpose of the expert system. For example; in an expert system which diagnosises faults, a final conclusion may be *fault_in_ft39,* or for an expert system which determines if a person is eligible for citizenship, a final conclusion may be *eligible_for_citizenship*. These facts are of use to the user.

There are many occasions when the answer wanted is not a simple *yes* or *no,* but is far more complex. In these cases a final conclusion is not as useful and the *print* statement is the better alternative. Deciding which method should be used to present advice to the user is dependent on the nature of the advice, and only a knowledge engineer can make that decision.

The form of conclusions in Aristotle are:

- <name> = <cond> <value>
- partial <name> = <cond> <value>

is a *partial* conclusion.

- final <name> = <cond> <value>

is a *final* conclusion. <cond> is defined as

addmem
<null>

For example:

spectrum_colour = addmem blue
partial inconsistent_flow = t
final eligible_citizen = t

At the end of a consultation the final and partial conclusions are printed out using templates and explanation strings. This is done in such a way that the distinction between the two can be seen. If there are no such conclusions, then the user is informed that there are no final or partial conclusions.

Partial Rules

The partial rule is an optional rule in every module, which if present, is always attempted after the goal rule has been attempted. The partial rule is like the goal rule in almost all aspects, the only difference is the fact that this rule is not used in backward chaining. Once a partial rule has given a value to a fact, then that fact can be used anywhere scoping allows.

Any rule can generate a partial conclusion, including a partial rule. This is done by indicating that the conclusion is partial.

if <condition> then ... partial p_sensor_faulty = T

A partial conclusion is identical to a normal conclusion and can be used wherever a normal conclusions is used. However a copy is made of the conclusion, which is used at the termination of the enquiry session.

Figure 4.8 illustrates how partial rules fit into the module structure of Aristotle. They are declared straight after the parameter declarations.

```
Module pipe ( param flow_in, flow_out, flow_meter ):

partial: if (flow_in <> flow_out) cor
            ((flow_in <> flow_meter) cor
             (flow_meter <> flow_out))
         then pipe_problem = T;
```

Figure 4.8 An example of a partial rule in a module.

If either *flow_in*, *flow_out* or *flow_meter* provide *unknown* as a result, then there can be no result. This partial rule will however still conclude that there is a problem with the pipe.

This chapter has explained the novel aspects of Aristotle's design and has shown how modules are structured, partial rules, normal rules and final rules are written and parameters used. The concepts of a parameter trail and search order are also explained.

BNF Definition of Aristotle

This is the BNF definition of Aristotle. The words in capitals are terminals in the language, for example; MODULE. Those in angle brackets are non terminals, for example; <Interface Section>, while those in lower case, without angle brackets, are pseudo terminals. The pseudo terminals are: *name,* which is any identifier with the first character being alphabetic, *number,* which is any integer, and *empty,* which is the empty production.

<Main Module> ::= MODULE MAIN : <Partial Condition>
<Interface Body> ENDMOD ;

<Interface Body> ::= INTERFACE <Interface Section>

<Interface Section> ::= <Interface Section> <Interface>
| <Interface>

<Interface> ::= <Fact Decl>
| <Module Decl>
| <Module Instant>

<Fact Decl> : FACT name : <Fact body>

<Fact Body> ::= <Typedecl> <Valuedecl> <Statusdecl> <Questiondecl>
<Explndecl>

<Typedecl> ::= TYPE : <Typetype> ;

<Typetype> ::= LOGICAL
| MEMBER
| NUMBER

<Valuedecl> ::= VALUE : <Valuetype> ;
| empty

<Valuetype> ::= <Member set>
| number .. number
| empty

<Statusdecl> := STATUS : <Statustype> ;

<Statustype> ::= ASK

| INFER
| BOTH
| PARAM

<Questiondecl> ::= QUESTION : string ;
| empty

<Explndecl> ::= EXPLN : string ;
| empty

<Module decl> ::= MODULE name <Param Decl> : <Partial Condition>
<Interface Body> <Body> ENDMOD ;

<Param decl> ::= (PARAM <Param Section>)
| empty

<Param Section> ::= <Param Section> , name
| name

<Module Instant> ::= name USE name <Argument Decl> : string ;

<Argument Decl> ::= (ARGUMENT <Arg Section>)
| empty

<Arg Section> ::= <Arg Section> ; <Argument>
| <Argument>

<Argument> ::= VAR name = name
| VAL name = <Condition>

<Partial Condition> ::= PARTIAL : IF <Condition> THEN <Conclusion> ;
| empty

<body> ::= GOAL : <Goal Rule> <Rule Section>

<Goal Rule> ::= IF <Condition> THEN <Conclusion> : string ;

<Rule Section> ::= <Rule Section> <Rule>
| empty

<Rule> ::= number : IF <Condition> THEN <Conclusion> : string ;

<Conclusion> ::= <Conclusion> & <Conclusion>
| <Conclusion Elt>

<Conclusion Elt> ::= name = <Condition>
| name = ADDMEM name
| PARTIAL name = <Condition>
| PARTIAL name = ADDMEM name
| FINAL name = <Condition>
| FINAL name = ADDMEM name
| PRINT string

<Condition> ::= <Condition> OR <Condition>
| <Condition> AND <Condition>
| <Condition> COR <Condition>
| <Condition> CAND <Condition>
| NOT <Condition>
| <Condelt>

<Condelt> ::= <Condelt> IS <Condelt>
| <Condelt> IN <Member Set>
| <Member Set> IN <Condelt>
| <Condelt> = <Condelt>
| <Condelt> > <Condelt>
| <Condelt> >= <Condelt>

| <Condelt> < <Condelt>
| <Condelt> <= <Condelt>
| <Condelt> <> <Condelt>
| <Condelt> BT <Condelt> AND <Condelt>
| <Condelt> NBT <Condelt> AND <Condelt>
| <Expr>

<Expr> ::= <Expr> + <Expr>
| <Expr> - <Expr>
| <Expr> / <Expr>
| <Expr> * <Expr>
| (<Condition>)
| <Ident>

<Ident> ::= number
| <tfdui>
| name

<tfdui> ::= t
| f
| d
| u
| i

<Member Set> ::= [<Member List>]

<Member List> ::= <Member List> , name
| name

CHAPTER 5 EXAMPLES AND DISCUSSION

This chapter will demonstrate how the techniques outlined in chapters 3 and 4 are an aid in reducing the complexity of a knowledge base and enable the expert system's knowledge to degrade gracefully and so provide a logic which more closely approximates a human expert's reasoning. It will show that by modularising the knowledge base, that base becomes less complex and so is easier to build and maintain, and will then show how modules and partial rules can be used to enable an expert system's knowledge to degrade gracefully. Finally the usefulness of the five value logic will be discussed by showing that it can handle the question of existence in a more natural fashion.

Three examples will be looked at; the first is a simple system of straight pipes with flow meters and T-junctions, the second is a simplified version of a real-life fault diagnosis expert system being built by the author for a local zinc processing plant to detect faults in a solution heater system, which will be used in our discussion on modules and partial rules and the third example is a simple expert system to check if a car's lights work and will be used to show the usefulness of the multi-value logic.

To demonstrate that one knowledge base is less complex than another, some metric to measure complexity is needed. Laufman (1987) proposes the number of rules and facts as a measure of the size of the knowledge base. In general this is not sufficient to measure complexity. For example; it does not take into account how the facts and rules are structured. Consider two examples; the first has ten rules and ten facts and complex expressions in the rules, the second has fifteen rules and fifteen facts but the expressions are less complex. Which knowledge base is the more complex? The answer depends on more than the number of rules and facts; there is a need to consider the complexity of the expressions themselves as the number of rules and facts only gives a guide to the size of the knowledge base, not necessarily its complexity.

Another useful metric is the number of references to facts in the knowledge base as compared to the number of facts declared. Consider two knowledge bases with number of references per fact of 1.55 and 1.83. The first knowledge base is referring to a fact,

on average, more often than the second knowledge base. The knowledge engineer of the second example may take the view that the more intermediate results obtained - the better the system - that is;

if A or B or C then Z
if E and F then A
if D and G and B then B
if I and A and H then C

whereas the knowledge engineer of the first expert system might have taken the view that the knowledge should be compact; that is:

if (E and F) or (D and G and E and F) or (I and E and F and H) then Z

In the first example there are 1.55 references per declared fact and in the second 1.83, but of course different domains may mean that the first is not as clear as the second and therefore, by itself, the number of references per fact are not completely satisfactory as a metric for measuring complexity.

Another useful metric is the number of references to facts as compared to the number of expressions in the knowledge base. The average number of references per expression will show how complex the expressions are in the knowledge base. A high number of references per expression shows that the expressions are long and therefore most likely complex, while a lower number of references per expression means that the expressions are smaller. With a linear expert system this is equivalent to the number of references per rule and in a modular expert system, since expressions can also be found in module parameters, it is the number of references per rule or module argument. Consider two examples;

if (A or B) then C
if (D and E) then A
if (F and G) then B

and

if (D and E) or (F and G) then C

In the first example there are 3.00 references per expression and 5.00 references per expression in the second example. The second example is a more complex rule than the three rules of the first example. Though again this can be taken to the extreme of having a large number or simple rules, which, because of the number, is harder to understand than a lesser number of more compact rules. So again this metric is not suitable by itself.

A further metric is the average number of rules, which the knowledge engineer has to look at, to find every reference to a particular fact. With linear expert systems this corresponds to the whole knowledge base and gives a guide as to how hard it is to modify a knowledge base. The knowledge engineer has to check for interactions between rules which are due to facts being used in more than one rule. This metric, while measuring the inter-actions, does not take into account how complex the expressions are in a rule.

Each of these metrics can only measure certain aspects of the complexity of a knowledge base, but if they are looked at together they can give a good guide to the complexity of a particular knowledge base. This thesis will look at all these metrics and use them as a means of comparing the complexity of the various knowledge bases.

Table 5.1 provides a comprehensive list of the number of rules, facts, expressions, references, distinct facts and rules searched for all the experiments discussed in this chapter. Table 5.2 shows the number of rules, facts, the average number of rules per fact, references per expression, references per fact and rules searched per distinct fact. These tables are a summary of the data to be found in Appendices C,D,H,I and J.

	Facts	Referencess	Distinct Facts	Expressions	Rules	Rules to Search
Exp 1. Linear	11	32	11	7	7	77
Exp 1. Modular	22	44	13	19	8	54
Exp 2. Linear	15	43	15	9	9	135
Exp 2. Modular	23	49	14	22	8	58
Exp 3. Linear	13	40	12	7	7	84
Exp 3. Modular	24	50	15	19	8	60
Exp 4. Linear	35	213	35	53	53	1855
Exp 4. Modular	56	179	43	80	32	784

Table 5.1 The statistics for Experiments 1, 2, 3 and 4

Example 1

This first example will centre around the following system of pipes. (see figure 5.1) It consists of a pipe with flow meter, a T-junction and another pipe with flow meter. There are two outputs and one input, and the flows are measured; and it will be assumed that these external flow meters are never faulty. This assumption is acceptable for this example because it is not desired to clutter the knowledge base up with too many rules at this stage.

	Rules	Facts	References per Expression	References per Fact	Rules Search'd per Dist. Fact
Exp 1. Linear	7	11	4.57	2.91	7.00
Exp 1. Modular	8	22	2.32	2.00	4.15
Exp 2. Linear	9	15	4.78	2.87	9.00
Exp 2. Modular	8	23	2.23	2.13	4.14
Exp 3. Linear	7	13	5.71	3.08	7.00
Exp 3. Modular	8	24	2.63	2.08	4.00
Exp 4. Linear	53	35	4.02	6.09	53.00
Exp 4. Modular	32	56	2.24	3.20	14.00

Table 5.2 Further statistics for Experiments 1, 2, 3 and 4

In figure 5.1 components A and C are identical. Each consists of a length of pipe with a flow meter in the middle (see figure 5.1a). The T-junction, component B (Figure 5.1b), splits the fluid into two streams according to the ratio, *Split Percentage*. The *Split Percentage* is the percentage of the fluid which is diverted away from the main stream and into the vertical part of the T junction. The full knowledge bases for the linear and modular versions can be seen in Appendix A.

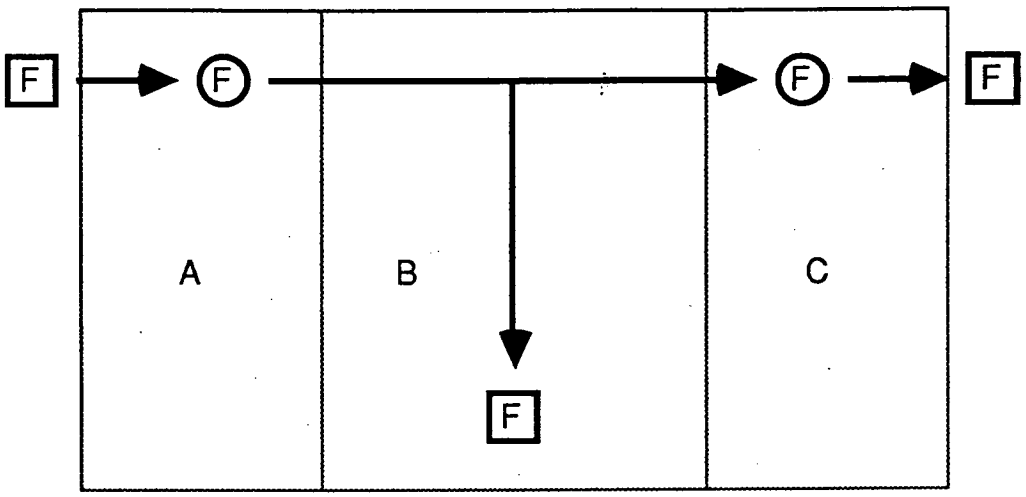


Figure 5.1 A system of Pipes with a T-Junction

Figure 5.1a A Pipe with one Flow Meter

Experiment 1

The first experiment will look at the system of pipes with four leaks (Figure 5.2). One leak of 10 units before the first flowmeter in Pipe 1, one leak of 10 units after the first flow meter and before the T-junction, another one of 20 units after the T-junction and before the flow meter in Pipe 2 and the last one of 10 units after the flow meter in Pipe 2. Figure 5.3 is a transcript of the session which detected these leaks.

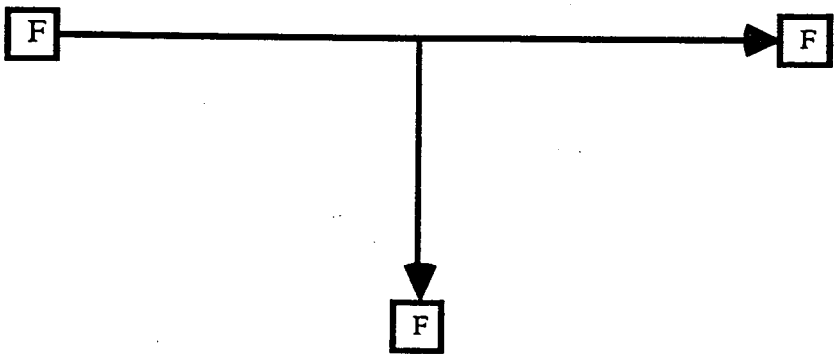


Figure 5.1b A T-Junction with no meters

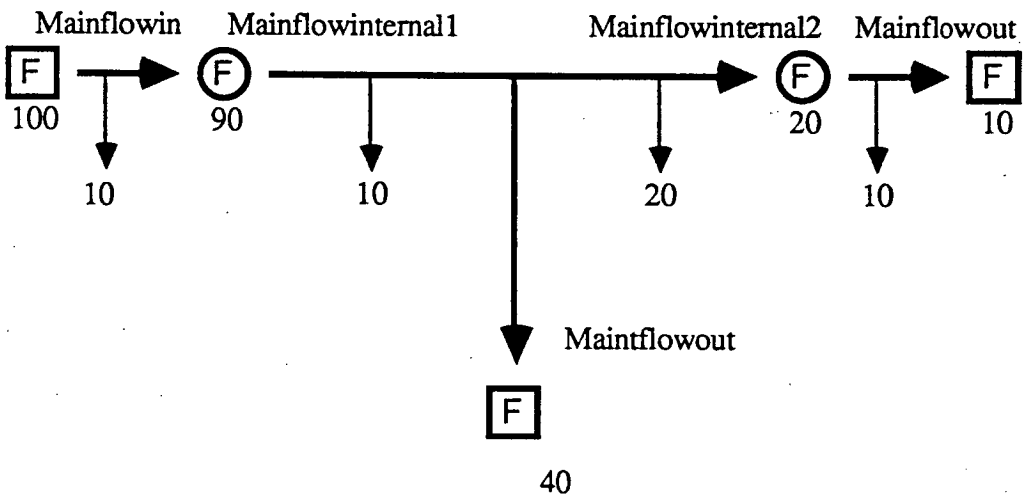


Figure 5.2 A pipe system with four leaks.

Figure 5.3 shows that there were five conclusions, the first and fourth concluded that there was a leak before the flow meter in Pipe 1 and after the flow meter in Pipe 2. The second concluded that there was a leak after the first flow meter, and the third and

```
In lineprob what is the input flow?
Values : Any number
Answer :  100.
In lineprob what is the internal flow in pipe 1?
Values : Any number
Answer :  90.
In lineprob what is the internal flow in pipe 2?
Values : Any number
Answer :  20.
In lineprob what is the flow from the T-junction?
Values : Any number
Answer :  40.
In lineprob what is the output flow?
Values : Any number
Answer :  10.
Result : t
In Pipe 1, Leak before flow meter has the value True.
In Pipe 1, Leak after flow meter has the value True.
In Pipe 2, Leak before flow meter has the value True.
In Pipe 2, Leak after flow meter has the value True.
In Junction 1, Leak in PTout pipe has the value True.
```

Figure 5.3 An example session with the leaks described in figure 5.2
The figures in bold have been entered by the user.

fifth concluded there was a leak in the PTout (T-junction) section of the pipe.

Two knowledge bases were constructed; one modular and one linear. The rules can be seen in Appendix A, while figure 5.4 shows their structure. It shows that in the modular version, there are two parts to the expert system. The first is the module *PGFP*, which is used for Pipe 1 and Pipe 2, and the second is module *PTOUT*, which is used for the T-Junction. Identifying errors in the knowledge base becomes easier, because

firstly, it is easier to locate rules and secondly, it is easier to find related rules. They are either in the current module or in a module, where the related facts are parameters. With the linear version, a rule can be anywhere in the knowledge base and can react with any other rule in the knowledge base.

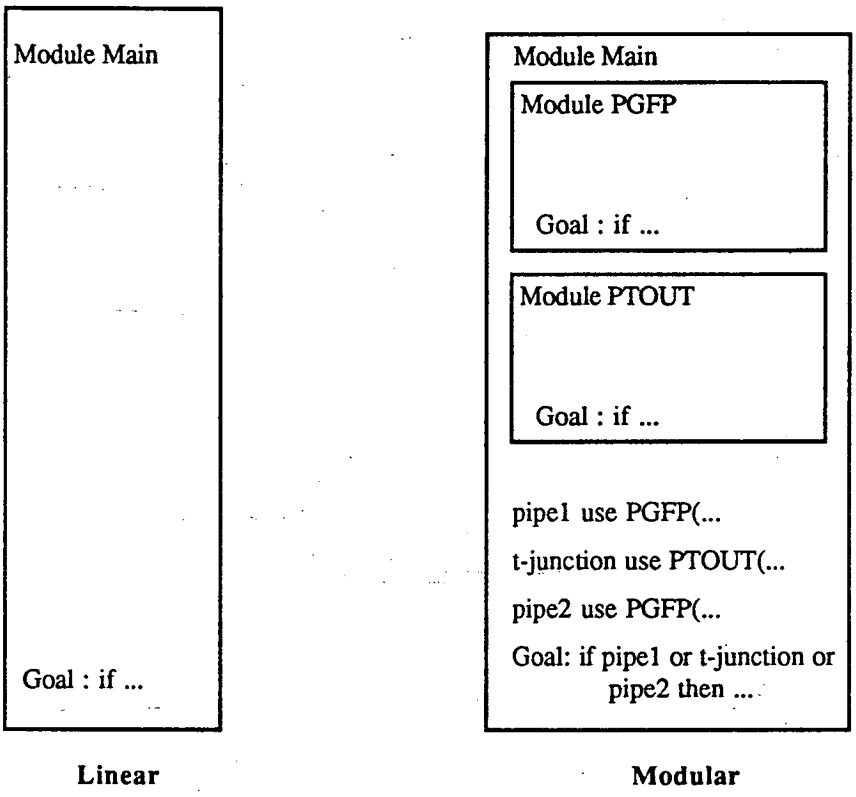


Figure 5.4 A comparision of the structure of the linear and modular knowledge bases of Appendix A.

Experiment 2

The second experiment modified the system to add an extra T-junction (see figure 5.5). The full knowledge bases can be found in Appendix B. On the T output pipe of the T-junction an extra T-junction is added. The structure of the knowledge bases are almost identical to that of experiment 1, with the exception that there is an added *use* statement in the modular version.

pipe1 use pgfp(...

t-junction1 use ptout(...

t-junction2 use ptout(...

pipe2 use pgfp(...

The structure of the linear knowledge base did not change, only its size. This was obviously due to the extra rules necessary for the additional T-junction.

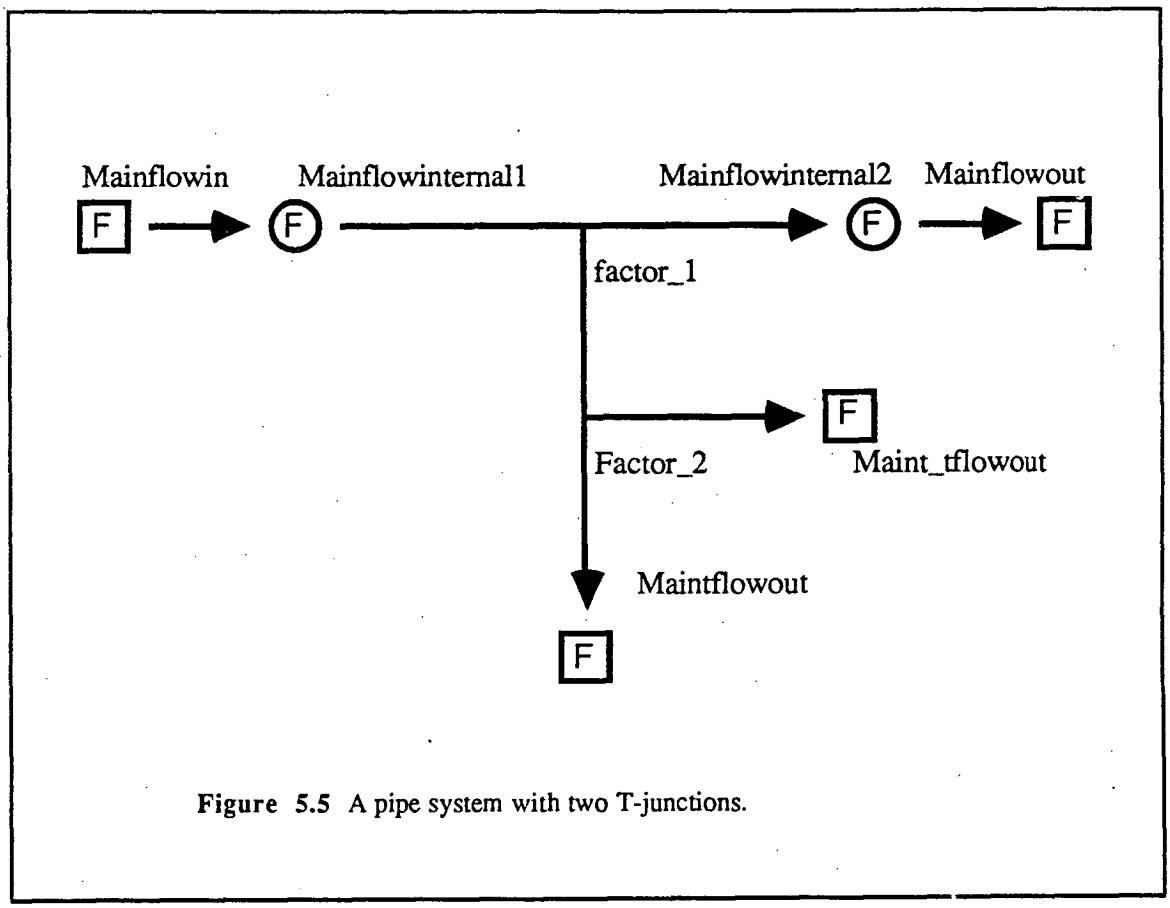


Figure 5.5 A pipe system with two T-junctions.

Experiment 3

For the third experiment, the knowledge base from experiment 1 was used with the following scenario. The input flow was 100 units, the output flow of the T-junction was 50 units and the output flow from the whole system was 50 units. The split percentage was 50%, but the flow meter in Pipe 2 read 70 units, and because the external

flow meters were assumed to always read true, then the meter in Pipe 2 must be faulty. (See figures 5.6 and 5.7.)

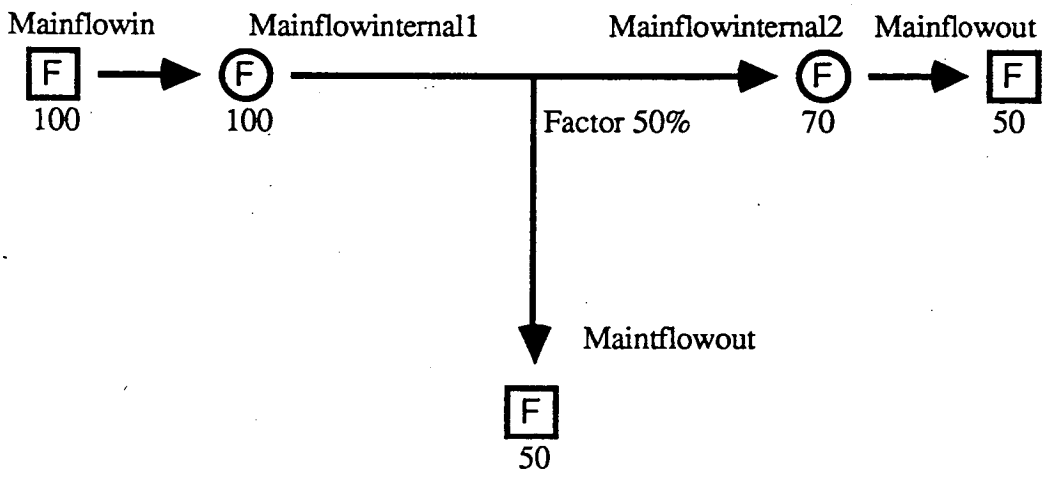


Figure 5.6 A pipe system with faulty internal meters.

The two versions from experiment 1, modular and linear, concluded that there was a leak after the flow meter in the second pipe. Such a conclusion was justifiable, if unacceptable, because that knowledge base did not know that an internal meter could be faulty. Those knowledge bases were then modified to compare the difficulties involved in modifying the knowledge bases, (see Appendix E). However, this expert system still cannot diagnose faulty sensors as part of its normal rules, but can as part of the partial rules. The partial rules introduced can be found in figure 5.8 while statistics can be seen in table 5.2

```
In lineprob what is the input flow?
Values : Any number
Answer :  100.
In lineprob what is the internal flow in pipe 1?
Values : Any number
Answer :  100.
In lineprob what is the internal flow in pipe 2?
Values : Any number
Answer :  70.
In lineprob what is the flow from the T-junction?
Values : Any number
Answer :  50.
In lineprob what is the output flow?
Values : Any number
Answer :  50.
Result : t
In Pipe 2, Leak after the meter has the value True.
also The problem with the line is True. However,
In Pipe 1, Likely the sensor faulty has the value True,
In Pipe 2, Likely the sensor faulty has the value True,
In Junction 1, Fault in T-junction has the value True,
nothing else is known.
```

Figure 5.7 An example session with the leaks described in figure 5.6 where the knowledge base has been modified to include partial knowledge. Knowledge of faulty sensors has not been added to the normal knowledge base.
The figures in bold have been entered by the user.

Example 2

The second example to be considered, was an expert system to diagnose faults which occur in solution heaters. The original expert system was written using Personal Consultant Plus[4] (PC+) on an IBM PS/2, and the knowledge base was simplified and transferred to Aristotle. The modifications were necessary for two reasons; firstly because the original version covered more than just solution heaters, and secondly because PC+ includes features which were not included in Aristotle. For example; trending functions were not put into Aristotle because they were added complications.

[4] Personal Consultant Plus (TM). Texas Instruments Ltd.

```
if balance is t cor notequal is t then perr = t;

10 : if (((flowin <> flowout) cor
          (flowout <> flowinternal)) cor
          (flowin <> flowinternal))
     then partial notequal = t :
          'The flows are not equal';

11 : if ((flowin < flowinternal) cor
          (flowinternal < flowout)) cor
          (flowin < flowout)
     then partial balance = t :
          'Likely the sensors are faulty';
```

Figure 5.8 The partial rule used in figure 5.7 and Appendix E. The partial rule uses two other rules, rules 10 and 11 as part of its reasoning process. Rules 10 and 11 are not used by any other rule in the knowledge base.

The domain being encoded consists of three steam solution heaters, of which only one is in use at any time. The solution flow into the heaters is not controlled, though it is measured. The solution is sent to one of the three heaters depending on which is selected, and is heated to a set temperature before being sent onto the next stage. The steam is controlled to maintain the temperature by a feed-back loop, and the input and output flows are known, as are the temperatures. There is a high-temperature trip, which if any one of the three heaters exceeds a pre-determined value, then the system closes down. A diagram of the section can be seen in figure 5.9 and figure 5.10 shows the structure of the knowledge base.

The knowledge base has one sub-module, called *Heater_System*. This sub-module contains all the knowledge relating to the heaters themselves and the splitting of the solution that goes to the selected heater. It has a sub-module itself, *Heater* which has three instantiations, one for each of the heater units. This module has the rules relating to faults in the heaters, such as leaks and blockages. The module *Heater_System* has rules to diagnose faults in the pipes leading to the heaters. This module contains

the knowledge for finding faults in the heaters, pipes and valves which divert the steam and solution to the selected heater, whilst the outer or main module *Main* encompasses all these modules, plus rules on the solution feed lines. The modular knowledge base can be found in Appendix F and the linear in Appendix G.

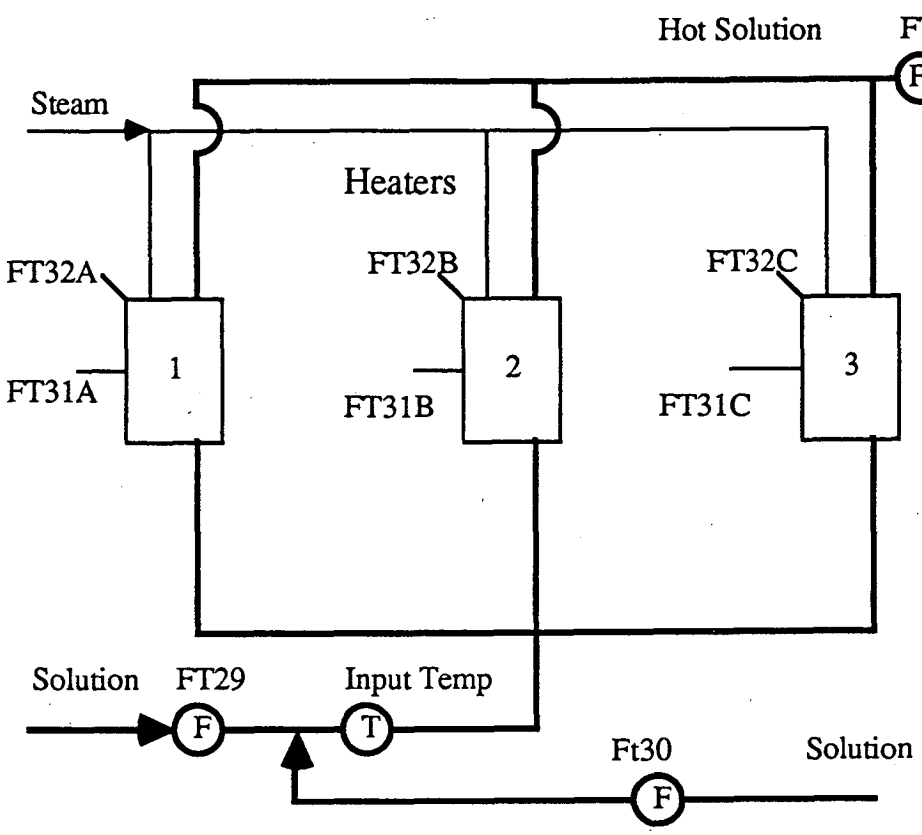


Figure 5.9 The Heater System of Appendices F and G

The expert system has to determine if there are any faults in the plant. These include:

- Leaks in steam and solution lines,
- Blockages in steam and solution lines,
- Sensor failures,

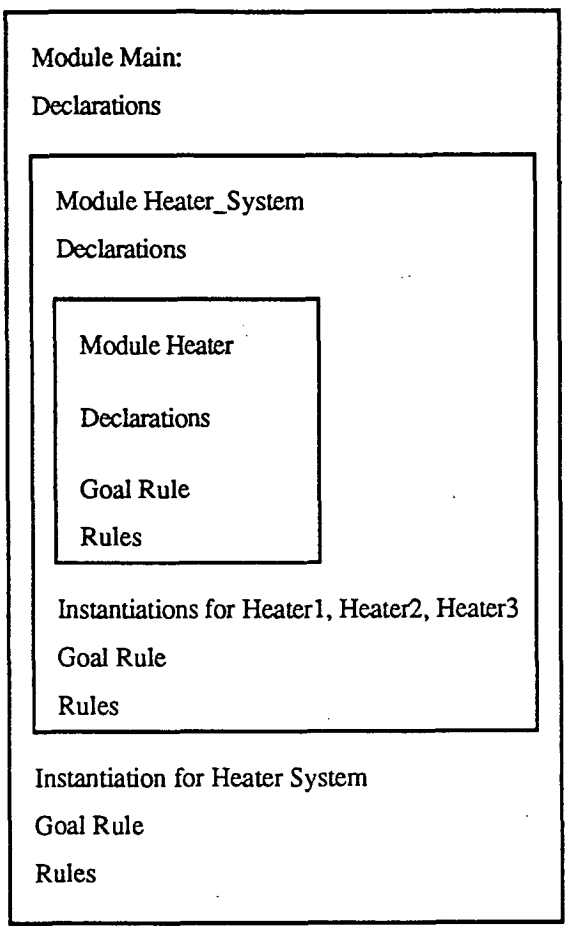


Figure 5.10 The structure of the Modular Heater System Expert System

- Valve failures and
- Miscellaneous faults.

One of the miscellaneous faults is an attempt to control the steam flow from the wrong heater. The original expert system is designed to start up when an alarm goes off, and to remain dormant otherwise.

Experiment 4

In the fourth experiment, there is a leak in the heater. (Figure 5.11, shown on page 75, is a cut down version of the transcript of the session, and the full version can be found in Appendix M.) The system is using Heater 1, with a solution input flow of *Flow Meter 29* + *Flow Meter 30* giving 30 units. The output flow of *Flow Meter 33* is 1 unit. The steam mass flow is 1 unit and the low reading for steam mass flow is 3. The expert system deduces that there is a leak in or before the Heater.

> **In Heater 1, LEAK in the HEATER System has the value True.**
> **also On a more general level,**
> **In Heater 1, Leak, Blockage or Faulty Sensor has the value True,**
> **nothing else is known.**

Here there are two conclusions, a final as well as a partial conclusion. During the consultation the expert system does print out a message giving the fault:

> **There is a leak before or in the HEATER**

In this example, two methods for providing advice to the users are mentioned, firstly, messages are printed during the consultation, and secondly, *final* conclusions are used.

Experiment 5

In the fifth experiment, data was given to the expert system so that it could not reach a conclusion. (The transcript can be found in Figure 5.12) The scenario was, that there was a faulty temperature gauge and the temperature half way along the heater was greater than the output temperature, but both values were within reasonable bounds. Because of the way the knowledge base was constructed, the expert system could not determine which of the two was faulty, and when the session was run, the expert system could not reach a decision, but using partial knowledge did conclude that:

```
In Heater System Controlling from which Heater 1 2 3?
Values :   1..3
Answer :  1.
In Main Module What is the Steam Mass Flow?
Values : Any number
Answer :  1.
In Heater System What is the flow for Flow Meter 29?
Values : Any number
Answer :  25.
     :
There is a leak before or in the HEATER
In Heater System What is the value of Temp. Meter 32A?
Values : Any number
Answer : 95.
In Heater System What is the value of Temp. Meter 31A?
Values : Any number
Answer : 90.
     :
     :
In Heater System Is the Middle Temp. in Heater 1 steady?
Values : t f d u i
Answer : t.

Result : t.

In Heater 1, LEAK in The HEATER has the value True.
also on a more general level,
In Heater 1, Leak, blockage or Faulty sensor is True.
Nothing else is known
```

Figure 5.11 An extract from the transcript of an enquiry session which was given data such that it could reach a decision The full transcript can be found in Appendix O.
The figures in bold have been entered by the user.

In Heater 1, Temperature Gauge Fault has the value True

This information, though it did not detail the fault exactly, did guide an operator to where the fault may have been.

Example 1 - Discussion

In the first experiment it is seen that the linear knowledge base is smaller than the modular knowledge base; 7 rules as compared with 8, and 11 facts as compared with 22. The number of expressions has increased from 7 to 19, and similarly the number of distinct facts has gone from 11 to 13. References have increased from 32 to 44, but the total number of rules, which need to be searched, has decreased from 77 to 54, and looking at the number of references per fact, we find that this has come down, on average, from 2.91 to 2.0. Similarly, the number of references per expression has shown a significant decrease, from 4.57 to 2.32, while the number of rules searched per distinct fact has decreased from 7.00 to 4.15.

The linear version of the knowledge base is smaller than the modular version. The number of rules and facts are smaller, as are the number of references, expressions and distinct facts. On the other hand, the modular version is *less* complex than the linear version, since the number of references per fact have decreased; meaning that a particular fact is not being used as many times in the modular version, and the important indicators of complexity, references per expression and rules searched per distinct fact, both show significant drops, with the modular as compared to the linear versions - differences of over 40% for both. From these results it is inferred that with the modular knowledge base, it is easier to find a reference to a fact and to make changes to the rules.

In the second experiment the complexity of the knowledge base is not so obvious. The linear knowledge base is smaller than the modular knowledge base. There are 9 rules as compared to 8, and 15 facts as compared to 23. The number of expressions has increased, from 9 to 22, and the number of references from 32 to 44, but the number of distinct facts has decreased from 15 to 14, while the total number of rules searched has decreased significantly from 135 to 58. Looking at the number of references per

```
In Heater System Controlling from which heater  1 2 3 ?
Values : 1..3
Answer : 1.
In The Main Module What is the Steam Mass Flow?
Values : Any number
Answer : 25.
In Heater System What is the value of Temperature Meter 31B?
Values : Any number
Answer : 30.
In Heater System What is the value of Temperature Meter 32B?
Values : Any number
Answer : 30.
In Heater System What is the value of Temperature Meter 31C?
Values : Any number
Answer : 30.
In Heater System What is the value of Temperature Meter 32C?
Values : Any number
Answer : 30.
In The Main Module What is the Flow for Flow Meter 29?
Values : Any number
Answer : 20.
In The Main Module What is the Flow for Flow Meter 30?
Values : Any number
Answer : 5.
In The Main Module What is the Flow for Flow Meter 33?
Values : Any number
Answer : 25.
In Heater System What is the value of Temperature Meter 32A?
Values : Any number
Answer : 90.
In Heater System What is the value of Temperature Meter 31A?
Values : Any number
Answer : 95.
In Heater System Is the Output Temperature in Heater 1 Steady?
Values : t f d u i
Answer : f.
In Heater System Is the Middle Temperature in Heater 1 Steady?
Values : t f d u i
Answer : f.
In The Main Module What is the Steam Flow?
Values : Any number
Answer : 500.
In The Main Module What is the Steam Flow Valve Opening?
Values : Any number
Answer : 49.
In The Main Module What is the Steam Pressure?
Values : Any number
Answer : 150.
In The Main Module What is the Cold Solution Temperature?
Values : Any number
Answer : 31.
In The Main Module What is the Cold Solution Flow?
Values : Any number
Answer : 133.
Result : f
Problem with in the Heater System is False. However,
In Heater 1, Temperature Gauges Faulty has the value True,
nothing else is known.
```

Figure 5.12 An enquiry session for a problem with the heaters. The figures in bold have been entered by the user

fact, this experiment shows, that this has come down on average from 2.87 to 2.13, and the number of references per expression has significantly decreased, from 4.78 to 2.23, while the number of rules searched per distinct fact has decreased from 9.00 to 4.14.

The pattern in these figures is very similar to the first experiment as they describe a similar size of knowledge base. It is interesting to compare the differences between the two knowledge bases and to look at the relative changes in complexity for the linear and modular knowledge bases.

For the modular knowledge bases it is seen that the number of rules have *not* increased, remaining steady at 8, which is to be expected, because no extra rules were introduced whilst there was only one extra module instantiation. This explains the increased number of rules searched, expressions, facts and distinct facts, but the number of facts and distinct facts has only increased by one, as opposed to to an increase of four for the linear knowledge bases. There is only one extra flow meter in the pipe system and no extra intermediate facts are needed. On the other hand, extra intermediate facts are needed with the linear knowledge bases.

The increase of 58 rules to be searched in the linear knowledge bases is significantly greater than the increase of 4 obtained with the modular knowledge bases, which in effect means that the number of rules searched per distinct fact has remained steady at approximately 4.14. The drop from 2.32 to 2.23 in references per expression does not indicate very much because the differences are so slight; on the other hand, the facts are working harder in the second modular knowledge base, because the number of references per fact have increased as opposed to the second linear knowledge base where they have decreased. This is because the number of references in the module instantiations are greater than the increase in the number of declarations. On the other hand, the number of facts have increased at a faster rate than the number of references, though overall, the linear knowledge base has shown a greater increase in the complexity metrics than the modular knowledge base, because to add another component means adding more rules, while with the modular knowledge base, only a further

instantiation is added.

It is useful to look at where the changes have occurred when modifying the knowledge base. For the linear version, the changes showed themselves actually in the rules. For example; the expression for the output flow of Pipe 1 is:

flowinternal2 + tflowout

which needed to be rewritten as:

flowinternal2 + tflowout + t_tflowout

wherever it occurred in the knowledge base.

The same had to be done with almost every expression relating to the output from pipe 1 input to pipe 2 and the input and output to the first T-junction plus the new T-junction itself. In the modular version, a new *use* statement was inserted and the modifications, like those above, were only done once when the modules were instantiated.

```
pipe1 use pgfp ( ...
                val flowout = (mainflowinternal2 + maintflowout +
                        maint_tflowout) ) :...
```

No other modifications were required for the output of Pipe 1, but similar modifications were required for each of the other pipes in the system. Concentrating the modifications in this way means that the chances of forgetting instances were reduced; and the smaller number of rules that need to be searched, meant that it was easier to, find those references, which had been forgotten.

In the third experiment the knowledge base was modified to include partial knowledge on meter faults and to tighten up the knowledge on leaks and blockages. This type of modification was different from the type discussed in experiment 2, where a component was added; whilst in this case the components were the same, but the rules were changed.

Comparing modular and linear versions of experiment 3, it was seen that the linear knowledge base was smaller than the modular knowledge base, while for experiments 1 and 2, they were very much the same. The important point to note was that the number of rules did not change in either the linear or modular knowledge bases. Looking at the differences between the number of references per fact, it was seen that for experiment 1 this was 0.91, for experiment 2 it was 0.74, while for experiment 3 it was 1.00 all in favour of the modular knowledge base. Similarly for rules searched per distinct fact, the differences were 2.85, 4.86 and 3.00, while for references per expression they were 2.25, 2.55 and 3.08. From these figures there was no clear trend as to which was the more complex experiment, 2 or 3; except to say that the differences were all in favour of the modular knowledge base.

Example 2 - Discussion

In this example the knowledge bases are far larger than those described in example 1. In the linear version there are 53 rules, 35 facts, 53 expressions, 213 references, 35 distinct facts and 1855 rules to search, to find every reference to every fact, but in the modular version, by comparison, there are 32 rules, 56 facts, 80 expressions, 179 references, 43 distinct facts and 784 rules to be searched. This means that there are 6.09 references per fact for the linear and only 3.20 for the modular version; less than half. This is the reverse of what was found in the earlier experiments, and indicates that a fact in the modular knowledge base is being worked less than a fact in the linear knowledge base. Furthermore, the number of references per expression have also decreased from 4.02 to 2.24 for the modular system and so the expressions are simpler in the modular version than they are in the linear version. In addition, the average number of rules searched per distinct fact have decreased from 53.00 to 14.00; a cut of over one third. This shows that the interactions between rules are fewer, and that it is easier to find every reference to a particular fact.

Examining these values overall, it is seen that the modular knowledge base is smaller and the number of rules are less, even though the number of facts are not. It is less complex because the average size of an expression is less, and the facts are not being used as much; as evidenced by the lower number of references per fact for the

modular knowledge base. It is particularly interesting to look at the number of rules which need to be searched per fact; 14.00 as compared to 53.00. This difference is significant and demonstrates amply that the modular knowledge base is less complex and easier to modify than the linear knowledge base.

This domain lends itself to encoding, using modules. Of course some domains are not so receptive and the advantages gained by using modules as against not using modules depends heavily on how the domain is broken up. A problem divided up unsuitably, can be more complex and harder to understand, than one which is designed using linear principles; the same problem occurs in software engineering.

By now it is clear that the choice of structure for a modular knowledge base is very important. Using the top-down approach to design, tempered by bottom-up considerations, there are two main ways in which the knowledge base of experiment 4 can be modularised. The first decision that needs to be made is to identify what components are involved, and this choice must be take into account the purpose of the expert system, and what it is meant to do. In this case, the main components are the heater units, as they are the only components which stand out as being recognisable by themselves. The obvious choice then is to make each component a module, and since the behaviour of each, when faulty, is the same, then one module instantiated three times is the natural way of representing them.

The next decision to be made is how to group the modules together. There are two main choices; put them into one module, which represents the remainder of the pipe system, or to group them together into one heater system, where this module selects one heater and handles the diversion of solution and the faults that arise from it; and this module is then put into another module, which feeds the solution to the heater system.

Each option has its advantages and disadvantages. The advantage of only having one sub-module, the first option, is that it does not divide the knowledge up excessively, but the disadvantages are that it is not as easy to alter the number of heaters or transfer

the group of heaters to another situation, and the diversion of solutions are not as compact. With the second option, the heater selection and diversion are included into one module and hence they are easy to alter, but the disadvantages are that there are more modules and the module boundaries are not as distinct. The second option is preferable because it means that the heaters can easily be transferred elsewhere. There are three modules:

Main,
Heater_System and
Heater.

Figure 5.9 shows the Heater System and figure 5.10 depicts the structure of the Modular Heater System. Appendices F and G contain the knowledge bases.

In the linear version, (Figure 5.13), it is seen that rules 21-29,210,211 are very similar to 31-39,310,311 and 41-49,410,411, and that each of these groups concern one of the heaters; the only difference between these three sets of rules lie in which of the heater's values they cover. Rules 21-29,210,211 cover the values for heater 1, 31-39,310,311 cover heater 2 and 41-49,410,411 cover heater 3. These rules have been compressed to form one parameterised module called *Heater*. About half the rules in the module *Heater_System* are concerned with how the solution is divided, providing *soln1in, soln2out* and *soln3in* with values, wherever needed, and are not significantly different from the linear knowledge base.

When looking at the linear knowledge base it is not easy to find out where the different parts are. Rule labels, though they are a guide, are not necessarily good; they do not have to be in any order, and rules close together physically or label-wise, do not have to be related to each other. For example; in Appendix G, the labels of those rules relating to heater 1 start with 2, while heater 2's rules start with 3 and heater 3's rules start with 4. This is only one knowledge engineer's convention, and there is no reason why rule 329 can not deal with steam flow to heater 1. In general then, the knowledge base becomes chaotic quite quickly.

```
21: if (select = 1) cand (soln1in > (soln1out + 2))
       cand (steam_mf < steam_mf_lo)
    then print 'There is a leak before or in the HEATER'
         & final leak = t :
         'There is a leak before or in the HEATER';

22: if (select = 1) cand (soln1in > (soln1out + 2))
       cand (steam_mf bt steam_mf_lo and shsteam_mf_hi)
    then print 'There is a leak after the HEATER'
         & final leak = t :
         'There is a leak after the HEATER';

23: if (select = 1) cand ((soln1in - soln1out) bt -2 2)
       cand (soln1in < soln1in_lo)
    then print 'There is a Blockage in the HEATER'
         & final blockage = t :
         'There is a Blockage in the HEATER';

24: if (ft32a > 50) cand (select <> 1))
    then print 'Controlling from the WRONG HEATER'
         & final gen_fault = t :
         'Controlling from the WRONG HEATER';

25: if (ft31a > 50) cand (select <> 1))
    then print 'Controlling from the WRONG HEATER'
         & final gen_fault = t :
         'Controlling from the WRONG HEATER';

26: if (ft32a <= 1) cand (select <> 1))
    then print 'Faulty output temperature meters'
         & final sensor_fault = t :
         'Faulty output temperature meters';
```

Figure 5.13 Some of the rules which were used to make up the modules, Heater1, Heater2 and Heater 3. There are three groups of 13 rules, of which we see an example here. The other groups just slightly change the fact names, for example, SOLN2IN or SOLN3IN.

Summary

Comparisons show that it is easier to modify a modular knowledge base than a linear knowledge base, but in some cases, especially where modularity does not fit naturally this may not be so. The same is true for normal programming, although on the whole it is clear that modular knowledge bases are simpler to understand, simpler to modify and hence simpler to build. Inter-actions are reduced because the average size of the scope (as measured by the number of rules which need to be searched) is reduced, and it is no longer the entire knowledge base. Because the scope is smaller, there is a better chance of finding inconsistent rules in the knowledge base.

Partial Rules

In experiment 3 the response of experiment 1 was tested with a ridiculous scenario (figure 5.6). It responded by saying that there was a leak after the flow meter in Pipe 2. To the human observer this was ridiculous, but it was the logical conclusion from the knowledge provided to the expert system. That was the result of the way the rules were constructed. Aristotle cannot make a *value* judgement; that is the human's role. To aid the user, Aristotle listed those partial conclusions which were satisfied. To the operator, seeing the message *Leak,* followed by the message *Faulty Sensors,* would have immediately told him that there may have been more to the situation than a simple leak.

The practical advantage of partial rules is most evident in experiment 3, where the expert system reaches a conclusion which to an operator, who has detailed knowledge of that section of the plant, is obviously wrong; but to an operator who has control over a much larger section of the plant, with many hundreds of variables, it might not be so obvious that such a conclusion was erroneous. This reduces the credibility of the expert system because the user will be less likely to trust it later on.

Aristotle, by giving the user the benefit of the extra conclusions, puts the human in the position of being able to make decisions, based on all the available information.

In the fifth experiment, Aristotle is presented with a situation of which it knows nothing. It concludes that it cannot find a definite problem, but that the operator should look at the temperatures gauges in Heater 1. This is analogous to one specialist referring a patient to another specialist; that is in this particular case, the technicians should be called to check the apparent malfunction of the equipment.

In the third experiment the partial rules provide a back up to the normal deductive process. These rules, being of a more general nature than normal rules, provide the user with a large picture. The extra information puts the user in the position where he may decide to disregard, what in hindsight, may prove to be bad advice. This also aids in detecting errors so that they may be fixed. In the fifth experiment, the expert system cannot diagnosise the fault at all; the problem either was not encoded, or was encoded incorrectly. Despite this, the expert system is still capable of providing the user with sufficient data to correct the fault, though not necessarily as quickly.

In summary, partial rules provide the user with that extra service which would be expected from the human expert. They can handle not only those situations, which were not catered for, but also enables the user to make a judgement on how suitable is the advice from the expert system. These properties mean that less time is wasted as a consequence of the user following dead-end leads, and errors in the knowledge base coming to light sooner, which makes for easier maintenance.

Parameters

Parameters provide the knowledge engineer with the tools necessary to pull out a pipe or any component, and use that component at some other point without having to duplicate the knowledge, which means less effort on the part of the knowledge engineer. An example of this is seen in Appendix A and B. In Appendix B a second T-junction is introduced into the knowledge base, and as the two T-junctions do not necessarily have the same *split-percentage,* a straight duplication of the knowledge is not possible. The modular version handles this by having a parameter, FACTOR, with value 50%, which contains the split-percentage. Appendix B, on the other hand, is

reworked so that the expression:

flowmeter1 / 2

is replaced by

flowmeter1 / 4

wherever it occurs in the knowledge base and is related to the output from the *T* in the second T-junction. Simplicity demands that this replace

flowmeter1 / (2*2)

even though, the flow through the *T* of the second junction is a half of the flow through the *T* of the first T-junction, which is half of the flow going through *flowmeter1*. The expression

flowmeter1 / 2

occurs in those rules which relate to the the output of the first T-junction. If the second flowmeter does not divide equally then this would not be a simple conversion. But of course, these are not the only modifications required; they only calculate the actual flow out of the T-junction, and there are other complex expressions in the knowledge base. Quite often, simplifications have to be made to the expression so that they are readable, but in this process the knowledge loses the clarity which is expected of knowledge based systems. For example; one rule has the expression:

t_flowout > (flowmeter1 / 4)

On the surface, 4 seems to be totally unrelated to the knowledge base and this makes maintenance difficult. If parameters are used, then this process would be much simplier, as it is only necessary to pass in the parameter *factor*. Such a system is flexible, as it allows the knowledge engineer to easily alter the knowledge base, and this would have the effect of calculating *(flowinternal1 / 4),* and is far more readable and sensible from the engineers point of view.

Example 3

This is a small expert system which checks if a motor car's lights will pass an imaginary Motor Registry test. The expert system asks; if the car's headlights, stop-lights and high stop lights and spot lights, if they exist, work. In this experiment the five value logic, introduced in chapter 3, is used, and the knowledge bases may be found in Appendices J and K.

Experiment 6

This experiment covers two enquiry sessions with a car, which does not have spot lights, but does have high stop lights. The transcripts can be seen in Figures 5.14 and 5.15. In the first session, the expert system uses the extra logic value *irrelevant,* while in the second session it does not. As a result of not using the logic value *irrelevant,* the expert system has to ask firstly, if the car has spot lights and secondly, if they work. Similar enquires have to be made concerning the high stop lights.

Discussion

In the first version, where *irrelevant* is used, a response of *i* is used to indicate that the car does not have spot lights or high stop lights. This is similar to the response *N/A* which is used so frequently when filling in forms. There is no need to have an extra question to determine if the car has spot lights or high stop lights. Take for example; the rule 4 from the first session:

```
In Main Module Do the Head Lights work when turned on?
Values : t f d u i
Answer :  t.
In Main Module Are the Head Lights Aligned properly?
Values : t f d u i
Answer :  t.
In Main Module Do the Spot Lights work when turned on?
Values : t f d u i
Answer :  i.
In Main Module Do the Tail Lights work?
Values : t f d u i
Answer :  t.
In Main Module Do the High Stop Lights work properly?
Values : t f d u i
Answer :  t.
Result : t
In Main Module, All the Lights work has the value True.
On a more general level,
nothing else is known.
```

Figure 5.14 An example session with the example in Experiment 3 using Irrelevant The figures in bold have been entered by the user.

3: if (spots_work is t) cand (spots_aligned is t) then spots_ok = t :
'Check if the spotlights work and aligned';

4: if spots_work is i then spots_ok = t :
'Spot_lights do not exist';

The user is first asked whether the car's spot lights work. A response of *t* or *f* means that the car did have spot lights and the reasoning progressed normally from there. On the other hand, *i* signifies that the car does not have spot lights at all. This is extra

```
In Main Module Do the Head Lights work when turned on?
Values : t f d u i
Answer :  t.
In Main Module Are the Head Lights Aligned properly?
Values : t f d u i
Answer :  t.
In Main Module Does the Car have Spot Lights?
Values : t f d u i
Answer :  f.
In Main Module Do the Tail Lights work?
Values : t f d u i
Answer :  t.
In Main Module Does the Car have High Spot Lights?
Values : t f d u i
Answer :  t.
In Main Module Do the High Stop Lights work Properly?
Values : t f d u i
Answer :  t.
Result : t
In Main Module, All the Lights work has the value True.
On a more general level,
nothing else is known.
```

Figure 5.15 An example session with the example in Experiment 3 withoutusing Irrelevant The figures in bold have been entered by the user.

information which the expert system is able to use without explicitly asking the user.

This knowledge can be made explicit in the knowledge base by rules of the form:

if spots_work is t then have_spots = t
if spots_work is f then have_spots = t
if spots_work is d then have_spots = t
if spots_work is u then have_spots = u
if spots_work is i then have_spots = f

This may seem a long way of finding out if a car has spot lights, but from the users point of view it is better.

During an enquiry session, it is now no longer necessary to always check if items exist. Consider for example, the high stop lights in figure 5.15, the expert system first asks if the car has high stop lights and then if they work. In figure 5.14 the expert system goes straight ahead and asks if the high stop lights work. If the answer is *true* then the expert system not only knows that they work, but also that they exist; if however, the response is *false* then the high stop lights do not work, but they exist.

Do-not-know cannot be used to signify non-existence, for the user may honestly not know, whether or not the high stop lights work. A response of false is incorrect because this would mean that the car would fail the test when in fact it should pass. Uncertain reasoning does not overcome the problem of no high stop lights, because it would then most probably conclude that the car's lights are acceptable, but with a lower level of certainity. In the case of a car whose lights work properly, this is not an acceptable conclusion from an *expert system.*

This shows how the use of *irrelevant* may mean that an expert system can reduce the number of questions which it has to ask, and how it can use its knowledge more effectively. It also shows the situation, where the use of *do-not-know* to mean non-existence, is not suitable, and how, if an expert system does use *do-not-know* this way, then peculiar results can occur.

CHAPTER 6 THE FUTURE

This thesis so far has discussed; modules, and shown how they are an aid in reducing the complexity of knowledge bases, partial rules and how they can be used to ensure that an expert system's knowledge degrades gracefully and finally looked at a five value logic, and shown how it can be used to better model human reasoning patterns. In all these discussions Aristotle was used for experimental purposes. In this chapter it will be shown that modules and partial rules have a wider application than just goal-driven expert systems, of which Aristotle is one, and show that they apply also to data driven systems. It will also be shown how the five-value logic can be modified to include better forms of uncertain reasoning and will conclude by looking at improvements which need to be made to Aristotle, to bring it up to a standard acceptable for commercial exploitation.

Modules

A start will be made by looking at modules and data driven expert systems. Complexity is as much a problem with forward chaining systems as it is with backward chaining systems. As shown, modules provide a mechanism to divide the knowledge into related groups in such a way that similar rules do not have to be duplicated, and interactions between rules, while allowed, are controlled, thus helping to control complexity.

The functionality of modules needs to be changed however. In data driven systems, modules do not return a result. Instead they are of a form which is so familiar in todays systems. Module references are made in the conclusion part of the rule:

19 : if <condition> then module_pipe,

so that if the <condition> is satisfied, then the rules in the module *module_pipe* are investigated. Each module can be parameterised, so that data can be passed between a module and the calling module. In data driven systems, if a fact does not have a value, the control system does not explicitly try to get a value for the undefined fact. A module would terminate when the *terminating condition* of the module has been satisfied, or all the rules of the module have been tried, whichever occurs first. Rule

19 would be marked as being tried, if all the rules in *module_pipe* have been marked as being tried, or the *terminating condition* has been satisfied. On the other hand, if only some of the rules have been marked, and the *terminating condition* is not satisfied, and no more rules can be applied because some facts are undefined, then rule 19 would not be marked as being tried. After data has been added to the knowledge base then the unsatisfied rules are retried, including in this case, rule 19. Providing one of the parameters has been changed in some way, new data is then accessible to the module. Naturally those rules in *module_pipe,* which have been marked, would not be retried. The effect of this strategy is to cut down the number of rules which are tried in each cycle. Parameters mean that interactions should be reduced, and finding all the references to a fact should be made easier.

Partial Rules

It is now appropriate to look at partial rules and discuss the role which they might play in a data driven expert system. In earlier discussions, it was shown that, if partial rules are to be part of an expert system, then the natural place to put them is within a module. If it is assumed that this argument still applies, and it will shown later that it does, the first point to be considered is whether or not partial rules have a different role from normal rules in data driven expert systems.

One way for a knowledge base to degrade gracefully, is by providing it with generalised knowledge of related fields of expertise, a feature which is independent of the control strategy used for reasoning. How then is this extra knowledge to be included in the knowledge base of a data driven expert system? The natural mechanism is the *rule.*

Having arrived at the position of saying that extra rules need to be encoded for this extra knowledge, which will be called *partial knowledge,* it is necessary to examine whether or not these extra rules should have any special privileges beyond normal rules. There are two important features associated with the earlier discussion of partial rules in Aristotle, firstly, the control system always tries the partial rule, and secondly,

this is done *after* the rest of the knowledge is considered. In a normal data driven system, the control system cannot guarantee that these extra rules will only be tried after the rest of the rules in a module have been tried. Therefore these extra rules must be treated differently, and are in fact *partial rules*.

It is now necessary to show that the natural place for partial rules is within a module. Each module contains all the rules relating to a particular component, or group of components. Generalised knowledge on features of a component should be close to the component concerned; and the natural place is the module for the component. Therefore the natural place for *partial rules* is inside a module.

There are two main ways in which partial rules can be included in data driven expert systems. Firstly, there may be a separate knowledge base for the partial rules, but as shown with goal driven systems this is not always desirable, because a knowledge engineer may want to use the facts concluded, in partial rules. The second alternative, which is the one used in goal driven expert systems, is to have one partial rule, which can use all the knowledge that is available at the time it is tried. This would seem to be the most useful, for the same reasons described earlier with Aristotle.

Logic

A five value logic was introduced in Chapter 3 to overcome several problems associated with using classical logic. When this was discussed earlier, insufficient attention was paid to the problem of how the logic should handle uncertain reasoning.

To date, there are three main ways in which classical logic can be adapted to handle uncertain data and rules. The three are:

- Probabilities,
- Certainity Factors and
- Fuzzy Logic.

Each of these techniques can be applied with varying degrees of difficulty to the logic, however before this can be done, it is necessary to remove the logical values *do-not-know* and *unknown,* for they were only introduced to provide a crude form of uncertain reasoning.

This leaves a three value logic *true, false* and *irrelevant.* Similar techniques can be used to fuzzify this logic, as were used to fuzzify classical logic. However complications arise when intergrating the value *irrelevant* with *true* and *false*. There are similar complications with Certainity Factors as it is no longer possible to just have one single number ranging from -1 to 1, as exists with most certainity schemes, used today. Probabilities also have the same problems. The idea of something perhaps being irrelevant, does cause some difficulties. *True* and *False* are not a serious problem, because work is currently being done on that.

Aristotle

As Aristotle stands today it is not ready for commercial exploitation. Work needs to be done to improve error detection (typing and semantic) and recovery. The run-time system also needs improvements, including the user-interface and explanation facilities. Each of these improvements are essential for a public system, but not for experimental use.

CHAPTER 7 CONCLUSIONS

This is a report on the uses of modular knowledge bases which degrade gracefully, and a logic which is of more practical value in the real world than the logics currently in use.

It shows how modules can reduce the complexity of an expert system and consequently reduce the amount of work required to build and maintain them. Parameters and parameter trails are discussed with emphasis on how they can be used to pass information between modules.

Partial Rules are introduced to provide extra knowledge to the expert system about fields related to the domain of expertise, and how by using this knowledge, the expert system's knowledge base can be made to degrade gracefully; a feature that serves to heighten the apparent intelligence of the expert system.

A five value logic with the logical values *true, false, do-not-know, unknown* and *irrelevant* is introduced and it is demonstrated, that with this logic, the expert system can be made *smarter* by being able to handle a response of *irrelevant* from the user, and make deductions from such responses. It further eliminates the need to ask questions about the existence of objects or concepts, which results in the user being asked fewer questions.

The report concludes with a discussion on how these ideas can be extended beyond just the goal driven expert system described here, as they are also applicable to data driven systems.

Each of these points provide mechanisms which can make expert systems easier to build and maintain. They are beneficial to the expert system industry in general and knowledge engineers in particular.

BIBLIOGRAPHY

Alvey, PL, Myers CD, and Greaves MF, "An Analysis of the Problems of Augmenting a Small Expert System," *Research and Development in Expert Systems*, pp. 61-72, Cambridge University Press, 1984.

Barber, EO, *Expert Systems Survey*, W #122, University of Exeter, Dept of Computer Science, 1984.

Buchanan, BG and Duda RO, "Principles of Rule-Based Expert Systems," *Advances in Computers*, vol. 22, pp. 163-216, 1983.

Campbell, M and Berliner H, "A Chess Program that Chunks," *Proceedings of the National Conference on Artificial Intelligence*, pp. 49-53, 1983.

Chase, WG and Simon HA, "The mind's eye in chess," *Visual Information Processing*, Academic Press, New York, 1973. (ed) WG Chase

Clancey, WJ, "The Advantages of Abstract Control Knowledge in Expert System Design," *Proceedings of the National Conference on Artificial Intelligence*, pp. 74-78, 1983.

Clancey, WJ, "Classification Problem Solving," *Proceedings of the National Conference on Artificial Intelligence*, pp. 49-55, 1984.

deGroot, AD, *Thought and choice in chess*, Mouton Co, Paris, 1965.

Duda, RO and Hart PE, and Nilsson NJ, "Subjective Bayesian methods for rule-based inference systems," *AFIPS*, vol. 45, pp. 1075-1082, 1976.

Fickas, S, "Design Issues in a Rule-Based System," *Communications of the Association for Computing Machinery*, vol. 28, pp. 208-215, 1985.

Fikes, R and Kehler T, "The Role of Frame-Based Representation in Reasoning," *Communications of the Association for Computing Machinery*, vol. 28, pp. 904-920, 1985.

Forgy, C, Gupta A, Newell A, and Wedig R, "Initial Assessment of Architectures for Production Systems," *Proceedings of the National Conference on Artificial Intelligence*, pp. 116-120, 1984.

Frost, P, "A Natural Language Interface for Expert Systems: System Architecture," *Alvey Project*, vol. IKBS 1/24/007, 1987.

Gaines, BR, "Foundations of Fuzzy Reasoning," *International Journal on Man Machine Studies*, vol. 8, pp. 623-668, 1976.

Gaines, BR, "Foundations of Fuzzy Reasoning," *Int. J. Man-Machine Studies*, vol. 8, pp. 623-668, 1976.

Gallanti, M, Gilardoni L, Guida G, and Stefanini A, "Exploiting Physical and Design Knowledge in the Diagnosis of Complex Industrial Systems," *European Conference on Artificial Intelligence*, pp. 335-349, 1986.

Gordon, J and Shortliffe EH, "A Method for Managing Evidential Reasoning in a Hierarchical Hypothesis Space," *Artificial Intelligence*, vol. 26, pp. 323-357, 1985.

Griesmer, A, Hong SJ, Karnaugh M, Kastner JK, Schor MI, Ennis RL, Klein DA, Milliken KR, and Van Woerkom HM, "YES/MVS: A Continuous Real Time Expert System," *Proceedings of the National Conference on Artificial Intelligence*, pp. 130-136, 1984.

Hayes, PJ, "The Naive Physics Manifesto," *Expert Systems in the Micro-Electronic Age*, Edinburgh University Press, 1979. Michie D

Hewett, J, "Commercial Expert Systems in North Amercia," *European Conference on Artificial Intelligence Vol II*, pp. 96-102, 1986.

Hunter, JRW, *Artificial Intelligence in Medicine: A TUTORIAL SURVEY*, AIMG-8, University of Sussex, Artificial Intelligence in Medicine Group, 1986.

Instruments, Texas, *Personal Consultant Plus: Reference Manual*, Texas Instruments, 1987.

Keen, MJR and Williams G, "Expert System Shells Come of Age," *Research and Development in Expert Systems*, pp. 13-22, Cambridge University Press, 1984.

Kelly, VE and Steinberg LI, "The CRITTER System: Analyzing Digital Circuits by Propogating Behaviors and Specifications," *Proceedings of the National Conference on Artificial Intelligence*, pp. 284-289, 1982.

Koukoulis, C, *A Frame-based Method for Fault Diagnosis*, pp. 160-174, 1985.

Lee, LG Lock, K Teh, and R Campanini, "An Expert Operator Guidance System for an Iron Ore Sinter Plant," *Proceedings of the Third Australian Conference on Applications of Expert Systems*, pp. 61-86, 1987.

Little, SE, "Expert Systems: Developing an Organisational Framework," *Occasional Paper* , vol. No 10, Griffith University, 1986.

Merry, M, "Expert Systems - some problems and opportunities," *Expert Systems 85*, pp. 1-8, Cambridge University Press, 1985.

Newell, A, "Production Systems: Models of Control Structures," *Visual Information Processing*, Academic Press, 1973. Chase W

Olson, JR and HH Rueter, "Extracting Expertise from experts: Methods fpr knowldge acquisition," *Expert Systems*, vol. 4, pp. 152-157, 1987.

Pang, GKH and MacFarlane AGJ, "An Expert Systems Approach to Computer-Aided Design of Multivariable Systems," *Lecture Notes in Control and Information Sciences*, vol. 89, Springer-Verlag, 1987.

Quinlan, JR, "Discovering rules by induction from large collections of examples," *Expert Systems in the Micro Electronic Age*, Edinburgh University Press, 1979. (Ed) Michie D

Quinlan, JR, Compton P, Horn KA, and Lazarus L, "Inductive Knowledge Acquisition: A Case Study," *Technical Report Series*, vol. 86.4, The New South Wales Institute of Technology, Sydney, 1986.

Quinlan, JR, "Induction of decision trees," *Machine Learning*, vol. 1, 1986.

Quinlan, JR, "the effect of noise on concept learning," *Machine Learning*, vol. 2, Morgan Kaufmann, 1986.

Reichgelt, H and van Harmlen F, "Relevant Criteria for Choosing an Inference Engine in Expert Systems," *Expert Systems 85*, Cambridge University Press, 1985. Merry M

Reichgelt, H and van Harmlen F, *Criteria for Choosing Representation Languages and Control Regimes for Expert Systems*, DAI #287, University of Edinburgh, Dept of Artificial Intelligence, 1986.

Reitman, JS, "Skilled perception in Go: Deducing memory structures from inter-response times," *Cognitive Psychology*, pp. 336-356, 1976.

Shafer, G, *A Mathematical Theory of Evidence*, Princeton University Press, Princeton NJ, 1976.

Shortliffe, EH and Buchanan BG, "A model of inexact reasoning in medicine," *Math. Biosci.*, vol. 23, pp. 351-379, 1975.

Stefik, M, Aikins J, Balzer R, Benoit J, Birnbaum L, Hayes-Roth F, and Sacerdoti E, "The Organisation of Expert Systems: A Tutorial," *Artificial Intelligence*, vol. 18, pp. 135-172, 1982.

Urquhart, A, "Many-Valued Logics," *Handbook of Philosophical Logic Vol III*, D Reidel Pub, 1983. Gabbay D, Guenthner F

White, AP, "Inference Deficiencies in Rule-Based Expert Systems," *Research and Development in Expert Systems*, Cambridge University Press, 1984. Bramer MA

Wielinga, BJ and Breuker JA, "Models of Expertise," *European Conference on Artificial Intelligence*, pp. 306-318, 1986.

Williams, GJ, *GEM A Micro-Computer Based Expert System for Geographic Domains*, TR-CS-86-01, Australian National University, Dept of Computer Science, 1986.

Young, R, "Production Systems for Modelling Known Cognition," *Expert Systems in the Micro-Electronic Age*, Edinburgh University Press, 1979. Michie D

Zadeh, LA, "Fuzzy Sets," *Information and Control*, vol. 8, pp. 338-353, 1965.

Zadeh, LA, Fu KS, Tanaka K, and Shimura, *Fuzzy Sets and their Applications to Cognitive and Decision Processes*, Academic Press, 1975.

Appendix A

Modular Version

```
module main :

interface
   fact leakinline :
      type : logical;
      value :;
      status : infer;
      question :;
      expln : 'Leak in the line';

module line :


interface

   fact err:
      type : logical;
      value :;
      status : infer;
      question :;
      expln : 'There is a fault in the system';

   fact mainflowin :
      type : number;
      value :;
      status : ask;
      question :'What is the input flow';
      expln : 'The input flow';

   fact mainflowinternal1 :
      type : number;
      value :;
      status : ask;
      question :'What is the internal flow in pipe 1';
      expln : 'The flow measured internally on pipe 1';

   fact mainflowinternal2 :
      type : number;
      value :;
      status : ask;
      question :'What is the internal flow in pipe 2';
      expln : 'The flow measured internally on pipe 2';

   fact mainflowout :
      type : number;
      value :;
      status : ask;
      question :'What is the output flow';
      expln : 'The flow coming out';

   fact maintflowout :
      type : number;
      value :;
      status : ask;
      question : 'What is the flow from the T-junction';
      expln : 'Flow from t-junction';

module ptout (param flowin,flowout,tflowout,factor) :

partial : if flowin <> (flowout + tflowout) then partial terr = t;
```

```
interface
   fact terr :
      type : logical;
      value :;
      status : infer;
      question :;
      expln : 'Fault in the T-junction';

   fact flowin :
      type : number;
      value :;
      status : param;
      question :;
      expln : 'Input flow to the T-junction';

   fact flowout :
      type : number;
      value :;
      status : param;
      question :;
      expln : 'Output flow from the T-junction';

   fact tflowout :
      type : number;
      value :;
      status : param;
      question :;
      expln : 'Flow from T junction';

   fact factor :
      type : number;
      value :;
      status : param;
      question :;
      expln : 'Diversion factor for T-junction';

   fact blockedpipe :
      type : logical;
      value :;
      status : infer;
      question :;
      expln : 'Blocked outlet pipe';

   fact blockedtpipe :
      type : logical;
      value :;
      status : infer;
      question :;
      expln : 'Blocked T-outlet pipe';

   fact leak :
      type : logical;
      value :;
      status : infer;
      question :;
      expln : 'Leak in PTout pipe';

   fact err :
      type : logical;
      value :;
      status : infer;
      question :;
      expln : 'Some error in the pipe';
```

```
goal : if (((blockedpipe is t) or(blockedtpipe is t)) or
          (leak is t)) then err = t :
          'If there is a blocked pipe or a leak then a problem';
   1 : if (flowin <> (flowout + tflowout)) then final leak =  t :
          'Is there a leak in the pipe';
   2 : if tflowout > ((factor * flowin)/10) then final blockedpipe =  t:
          'If the T-flow out is greater than it is supposed to be';
   3 : if flowout > (flowin - ((factor * flowin)/10))
       then final blockedtpipe =  t:
          'If the flow out is greater than it should be';
endmod;


module pgfp (param flowin,flowinternal,flowout) :

partial : if ((flowin <> flowout) cor (flowout <> flowinternal))
             cor (flowin <> flowinternal) then perr =  t;

interface
   fact perr :
      type : logical;
      status : infer;
      expln : 'A leak in the Pipe';

   fact flowin :
      type : number;
      value :;
      status : param;
      question :;
      expln : 'The input flow';

   fact flowinternal :
      type : number;
      value :;
      status : param;
      question :;
      expln : 'The flow measured internally';

   fact flowout :
      type : number;
      value :;
      status : param;
      question :;
      expln : 'The flow coming out';

   fact leak :
      type : logical;
      value :;
      status : infer;
      question :;
      expln : 'Leak in flow pipe';

   fact leakbflow :
      type : logical;
      value :;
      status : infer;
      question :;
      expln : 'Leak before flow meter';

   fact leakaflow :
      type : logical;
      value :;
      status : infer;
```

```
      question :;
      expln : 'Leak after flow meter';

goal : if (leakbflow is t)  or (leakaflow is t) then leak =  t :
          'There is a leak is before or after the flow meter';
   1 : if (flowin > flowinternal)
          and (flowin <> flowout) then final leakbflow =  t :
          'There is a leak before the flow meter';
   2 : if (flowinternal > flowout)
          and (flowin <> flowout) then final leakaflow =  t :
          'There is a leak after the flow meter';
endmod;

pipe1 use pgfp (argument var flowin = mainflowin;
                         var flowinternal = mainflowinternal1;
                         val flowout =  (mainflowinternal2 + maintflowout)) :
                         'Pipe 1';

tjunction use ptout (argument var flowin = mainflowinternal1;
                               var flowout = mainflowinternal2;
                               var tflowout = maintflowout;
                               val factor = 5) :
                               'Junction 1';

pipe2 use pgfp (argument val flowin = (mainflowinternal1 - maintflowout) ;
                         var flowinternal = mainflowinternal2;
                         var flowout = mainflowout) :
                         'Pipe 2';

goal : if ((pipe1) or (pipe2)) or (tjunction)
          then err = t :
          'There is an fault in this line ';

endmod;

lineprob use line : 'Line 1';

goal : if lineprob then leakinline = t :
          'There is a problem with the line';
endmod;
```

Linear Version

```
module main :

interface
   fact flowin :
      type : number;
      value :;
      status : ask;
      question :'What is the input flow';
      expln : 'The input flow';

   fact flowmeter1 :
      type : number;
      value :;
      status : ask;
      question :'What is the internal flow in pipe 1';
      expln : 'The flow measured internally on pipe 1';

   fact flowmeter2 :
```

```
      type : number;
      value :;
      status : ask;
      question :'What is the internal flow in pipe 2';
      expln : 'The flow measured internally on pipe 2';

   fact flowout :
      type : number;
      value :;
      status : ask;
      question :'What is the output flow';
      expln : 'The flow coming out';

   fact tflowout :
      type : number;
      value :;
      status : ask;
      question : 'What is the flow from the T-junction';
      expln : 'Flow from t-junction';

      fact leakbmeter1 :
         type : logical;
         status : infer;
         expln : 'Leak before meter 1';
      fact leakbmeter2 :
         type : logical;
         status : infer;
         expln : 'Leak before meter 2';
      fact leakameter1 :
         type : logical;
         status : infer;
         expln : 'Leak after meter 1';
      fact leakameter2 :
         type : logical;
         status : infer;
         expln : 'Leak after meter 2';
      fact blockedpipe :
         type : logical;
         status : infer;
         expln : 'Blocked pipe';
      fact blockedtpipe :
         type : logical;
         status : infer;
         expln : 'Blocked T-junction pipe';
      fact problem :
         type : logical;
         status : infer;
         expln : 'Problem with the pipe system';

goal : if (leakbmeter1 is t) or
          (leakbmeter2 is t) or
          (leakameter1 is t) or
          (leakameter2 is t) or
          (blockedpipe is t) or
          (blockedtpipe is t)
       then problem = t: 'There is a problem with the pipe';


   0 : if (flowmeter1 > (flowmeter2 + tflowout))
          then final  leakbmeter2 = t :
          'Is there a leak in the pipe before meter 2';
   1 : if (flowmeter1 > (flowmeter2 + tflowout))
          then final  leakameter1 = t :
          'Is there a leak after meter 1';
   2 : if tflowout > ((5 * flowmeter1)/10)
```

```
        then final blockedpipe = t:
        'If the T-flow out is greater than it is supposed to be';
  3 : if flowmeter2 > (flowmeter1 - ((5 * flowmeter1)/10))
        then final blockedtpipe = t:
        'If the flow out is greater than it should be';
  4 : if (flowin > flowmeter1)
        and (flowin <> flowout) then final leakbmeter1 = t :
        'There is a leak before the flow meter';
  5 : if (flowmeter2 > flowout)
        and (flowin <> flowout) then final leakameter2 = t :
        'There is a leak after the flow meter';

endmod;
```

Appendix B

Modular Version

```
module main :


interface
   fact err:
      type : logical;
      value :;
      status : infer;
      question :;
      expln : 'There is a fault in the system';

   fact mainflowin :
      type : number;
      value :;
      status : ask;
      question :'What is the input flow';
      expln : 'The input flow';

   fact mainflowinternal1 :
      type : number;
      value :;
      status : ask;
      question :'What is the internal flow in pipe 1';
      expln : 'The flow measured internally on pipe 1';

   fact mainflowinternal2 :
      type : number;
      value :;
      status : ask;
      question :'What is the internal flow in pipe 2';
      expln : 'The flow measured internally on pipe 2';

   fact mainflowout :
      type : number;
      value :;
      status : ask;
      question :'What is the output flow';
      expln : 'The flow coming out';

   fact maintflowout :
      type : number;
      value :;
      status : ask;
      question : 'What is the flow from the T-junction';
      expln : 'Flow from t-junction';

   fact maint_tflowout :
      type : number;
      value :;
      status : ask;
      question : 'What is the flow from the second T-junction';
      expln : 'Flow from the second t-junction';

module ptout (param flowin,flowout,tflowout,factor) :

partial : if flowin <> (flowout + tflowout) then partial terr =  t;

interface
   fact terr :
```

```
      type : logical;
      value :;
      status : infer;
      question :;
      expln : 'Fault in the T-junction';

   fact flowin :
      type : number;
      value :;
      status : param;
      question :;
      expln : 'Input flow to the T-junction';

   fact flowout :
      type : number;
      value :;
      status : param;
      question :;
      expln : 'Output flow from the T-junction';

   fact tflowout :
      type : number;
      value :;
      status : param;
      question :;
      expln : 'Flow from T junction';

   fact factor :
      type : number;
      value :;
      status : param;
      question :;
      expln : 'Diversion factor for T-junction';

   fact blockedpipe :
      type : logical;
      value :;
      status : infer;
      question :;
      expln : 'Blocked outlet pipe';

   fact blockedtpipe :
      type : logical;
      value :;
      status : infer;
      question :;
      expln : 'Blocked T-outlet pipe';

   fact leak :
      type : logical;
      value :;
      status : infer;
      question :;
      expln : 'Leak in PTout pipe';

   fact err :
      type : logical;
      status : infer;
      expln :'Leak in T-junction';


goal : if (((blockedpipe is t) or(blockedtpipe is t)) or
           (leak is t)) then err =  t :
           'If there is a blocked pipe or a leak then a problem';
   1 : if (flowin <> (flowout + tflowout)) then final leak =  t :
```

```
          'Is there a leak in the pipe';
   2 : if tflowout > ((factor * flowin)/10) then final blockedpipe =  t:
          'If the T-flow out is greater than it is supposed to be';
   3 : if flowout > (flowin - ((factor * flowin)/10))
       then final blockedtpipe =  t:
          'If the flow out is greater than it should be';
endmod;



module pgfp (param flowin,flowinternal,flowout) :

partial : if ((flowin <> flowout) cor (flowout <> flowinternal))
             cor (flowin <> flowinternal) then perr =  t;

interface
   fact perr :
      type : logical;
      status : infer;
      expln : 'A leak in the Pipe';

   fact flowin :
      type : number;
      value :;
      status : param;
      question :;
      expln : 'The input flow';

   fact flowinternal :
      type : number;
      value :;
      status : param;
      question :;
      expln : 'The flow measured internally';

   fact flowout :
      type : number;
      value :;
      status : param;
      question :;
      expln : 'The flow coming out';

   fact leak :
      type : logical;
      value :;
      status : infer;
      question :;
      expln : 'Leak in flow pipe';

   fact leakbflow :
      type : logical;
      value :;
      status : infer;
      question :;
      expln : 'Leak before flow meter';

   fact leakaflow :
      type : logical;
      value :;
      status : infer;
      question :;
      expln : 'Leak after flow meter';

goal : if (leakbflow is t)  or (leakaflow is t) then leak =  t:
          'There is a leak is before or after the flow meter';
```

```
   1 : if (flowin > flowinternal)
         and (flowin <> flowout) then final leakbflow =  t :
         'There is a leak before the flow meter';
   2 : if (flowinternal > flowout)
         and (flowin <> flowout) then final leakaflow =  t :
         'There is a leak after the flow meter';
endmod;

pipe1 use pgfp (argument var flowin = mainflowin;
                         var flowinternal = mainflowinternal1;
                         val flowout = (mainflowinternal2 + maintflowout +
                                        maint_tflowout)) :
                         'Pipe 1';

tjunction_1 use ptout (argument var flowin = mainflowinternal1;
                                 var flowout = mainflowinternal2;
                                 val tflowout = ( maintflowout + maint_tflowout);
                                 val factor = 5) :
                                'Junction 1';

tjunction_2 use ptout (argument val flowin = (mainflowinternal1 -
                                              mainflowinternal2);
                                 var flowout = maintflowout;
                                 var tflowout = maint_tflowout;
                                 val factor = 5) :
                                 'Junction 2';

pipe2 use pgfp (argument val flowin = (mainflowinternal1 -
                                       (maintflowout + maint_tflowout)) ;
                         var flowinternal = mainflowinternal2;
                         var flowout = mainflowout) :
                         'Pipe 2';

goal : if ((pipe1) or (pipe2)) or ((tjunction_1)
          or (tjunction_2))
       then err =  t :
       'There is an fault in this line ';

endmod;
```

Linear Version

```
module main :

interface
   fact flowin :
      type : number;
      value :;
      status : ask;
      question :'What is the input flow';
      expln : 'The input flow';

   fact flowmeter1 :
      type : number;
      value :;
      status : ask;
      question :'What is the internal flow in pipe 1';
      expln : 'The flow measured internally on pipe 1';

   fact flowmeter2 :
      type : number;
```

```
      value :;
      status : ask;
      question :'What is the internal flow in pipe 2';
      expln : 'The flow measured internally on pipe 2';

   fact flowout :
      type : number;
      value :;
      status : ask;
      question :'What is the output flow';
      expln : 'The flow coming out';

   fact tflowout :
      type : number;
      value :;
      status : ask;
      question : 'What is the flow from junctionon 2';
      expln : 'Flow from junction 2';

   fact t_tflowout :
      type : number;
      value :;
      status : ask;
      question : 'What is the flow from the T-junctiono 2';
      expln : 'Flow from t-junction 2';

   fact leakbmeter1 :
      type : logical;
      status : infer;
      expln : 'Leak before meter 1';
   fact leakbmeter2 :
      type : logical;
      status : infer;
      expln : 'Leak before meter 2';
   fact leakameter1 :
      type : logical;
      status : infer;
      expln : 'Leak after meter 1';
   fact leakameter2 :
      type : logical;
      status : infer;
      expln : 'Leak after meter 2';
   fact blockedpipe1 :
      type : logical;
      status : infer;
      expln : 'Blocked pipe in 1';
   fact blockedtpipe1 :
      type : logical;
      status : infer;
      expln : 'Blocked T-junction pipe in 1';
   fact blockedpipe2 :
      type : logical;
      status : infer;
      expln : 'Blocked pipe in 2';
   fact blockedtpipe2 :
      type : logical;
      status : infer;
      expln : 'Blocked T-junction pipe in 2';
   fact problem :
      type : logical;
      status : infer;
      expln : 'Problem with the pipe system';

goal : if (leakbmeter1 is t) or
          (leakbmeter2 is t) or
```

```
          (leakameter1 is t) or
          (leakameter2 is t) or
          (blockedpipe1 is t) or
          (blockedtpipe1 is t)or
          (blockedpipe2 is t) or
          (blockedtpipe2  is t)
       then problem =  t: 'There is a problem with the pipe';


   0 : if (flowmeter1 > (flowmeter2 + tflowout + t_tflowout))
          then final  leakameter1 =  t :
          'Is there a leak after meter 1';
   1 : if (flowmeter1 > (flowmeter2 + tflowout + t_tflowout))
          then final  leakbmeter2 =  t :
          'Is there a leak before meter 2';
   2 : if (tflowout + t_tflowout) > ((5 * flowmeter1)/10)
          then final blockedpipe1 =  t:
          'If the T-flow out is greater than it is supposed to be';
   3 : if flowmeter2 > (flowmeter1 - ((5 * flowmeter1)/10))
          then final blockedtpipe1 =  t:
          'If the flow out is greater than it should be';
   4 : if t_tflowout > (flowmeter1 / 4)
          then final blockedpipe2 =  t :
          'If the T-flow out of junction 2 is greater than it should be';
   5 : if tflowout > (flowmeter1 / 4)
           then final blockedtpipe2 =  t :
           'If the flow out of the straight id more than it should be';
   6 : if (flowin > flowmeter1)
          and (flowin <> flowout) then final leakbmeter1 =  t :
          'There is a leak before the flow meter';
   7 : if (flowmeter2 > flowout)
          and (flowin <> flowout) then final leakameter2 =  t :
          'There is a leak after the flow meter';

endmod;
```

Table 1 Modular version
Number of references to facts and parameters and the number of rules where a fact can appear. (Appendix A)

Facts Name	No Refs	No Rules
Module Line		
factor	4	4
err	1	1
mainflowin	4	4
mainflowinternal1	10	8
mainflowinternal2	5	8
mainflowout	5	4
maintflowout	4	4
Module PGFP		
leakbflow	2	3
leakaflow	2	3
leak	1	3
Module PTOUT		
blockedpipe	2	4
blockedtpipe	2	4
leak	2	4
err	1	4

Table 2 Linear version
Number of references to facts and the number of rules where a fact could appear.

Facts Name	No Refs	No Rules
flowin	3	7
flowmeter1	6	7
flowmeter2	4	7
flowout	3	7
tflowout	3	7
leakbmeter1	2	7
leakbmeter2	2	7
leakameter1	2	7
leakameter2	2	7
blockedpipe	2	7
blockedtpipe	2	7

Table 1 Modular version
Number of references to facts and parameters and the number of rules where a fact can appear. (Appendix B)

Facts Name	No Refs	No Rules
Module Main		
err	1	1
mainflowin	4	4
mainflowinternal1	8	8
mainflowinternal2	6	8
maintflowout	4	4
maint_tflowout	3	4
mainflowout	3	4
Module PGFP		
leakbflow	2	3
leakaflow	2	3
leak	1	3
Module PTOUT		
blockedpipe	2	4
blockedtpipe	2	4
leak	2	4
err	1	4

Table 2 Linear version
Number of references to facts and the number of rules where a fact could appear.

Facts Name	No Refs	No Rules
flowin	3	8
flowmeter1	8	8
flowmeter2	4	8
flowout	3	8
t_flowout	4	8
t_tflowout	4	8
leakbmeter1	2	8
leakbmeter2	2	8
leakameter1	2	8
leakameter2	2	8
blockedpipe1	2	8
blockedtpipe1	2	8
blockedpipe2	2	8
blockedtpipe2	2	8
problem	1	8

Appendix E

Modular Version

```
module main :

interface
   fact leakinline :
      type : logical;
      status : infer;
      expln : 'A leak in the line';

module line (param factor) :


interface

   fact factor :
      type : number;
      value :;
      status : param;
      question :;
      expln : 'Split ratio for T-junction';

   fact err:
      type : logical;
      value :;
      status : infer;
      question :;
      expln : 'There is a fault in the system';

   fact mainflowin :
      type : number;
      value :;
      status : ask;
      question :'What is the input flow';
      expln : 'The input flow';

   fact mainflowinternal1 :
      type : number;
      value :;
      status : ask;
      question :'What is the internal flow in pipe 1';
      expln : 'The flow measured internally on pipe 1';

   fact mainflowinternal2 :
      type : number;
      value :;
      status : ask;
      question :'What is the internal flow in pipe 2';
      expln : 'The flow measured internally on pipe 2';

   fact mainflowout :
      type : number;
      value :;
      status : ask;
      question :'What is the output flow';
      expln : 'The flow coming out';

   fact maintflowout :
      type : number;
      value :;
      status : ask;
      question : 'What is the flow from the T-junction';
```

```
      expln : 'Flow from t-junction';

module ptout (param flowin,flowout,tflowout,factor) :

partial : if (flowin <> (flowout + tflowout)) cor
              (tflowout <> ((flowin * factor) /10)) cor
              (flowout <> (flowin - ((flowin * factor)/10)))
              then partial terr =  t;

interface
   fact terr :
      type : logical;
      value :;
      status : infer;
      question :;
      expln : 'Fault in the T-junction';

   fact flowin :
      type : number;
      value :;
      status : param;
      question :;
      expln : 'Input flow to the T-junction';

   fact flowout :
      type : number;
      value :;
      status : param;
      question :;
      expln : 'Output flow from the T-junction';

   fact tflowout :
      type : number;
      value :;
      status : param;
      question :;
      expln : 'Flow from T junction';

   fact factor :
      type : number;
      value :;
      status : param;
      question :;
      expln : 'Diversion factor for T-junction';

   fact blockedpipe :
      type : logical;
      value :;
      status : infer;
      question :;
      expln : 'Blocked outlet pipe';

   fact blockedtpipe :
      type : logical;
      value :;
      status : infer;
      question :;
      expln : 'Blocked T-outlet pipe';

   fact leak :
      type : logical;
      value :;
      status : infer;
      question :;
      expln : 'Leak in PTout pipe';
```

```
   fact err :
      type : logical;
      value :;
      status : infer;
      question :;
      expln : 'Some error in the pipe';


goal : if (((blockedpipe is t) or(blockedtpipe is t)) or
           (leak is t)) then err =  t :
           'If there is a blocked pipe or a leak then a problem';
   1 : if (flowin > (flowout + tflowout)) then final leak =  t :
         'Is there a leak in the pipe';
   2 : if (tflowout > ((factor * flowin)/10)) cand
         (flowin >= (flowout + tflowout))
         then final blockedpipe =  t:
         'If the T-flow out is greater than it is supposed to be';
   3 : if (flowout > (flowin - ((factor * flowin)/10))) cand
         (flowin >= (flowout + tflowout))
         then final blockedtpipe =  t:
         'If the flow out is greater than it should be';
endmod;



module pgfp (param flowin,flowinternal,flowout) :

partial : if balance is t cor notequal is t
          then perr =  t;


interface
   fact perr :
      type : logical;
      status : infer;
      expln : 'A fault in the Pipe or sensor';

   fact balance :
      type : logical;
      status : infer;
      expln : 'The sensors are most likely faulty';

   fact notequal :
      type : logical;
      status : infer;
      expln : 'There is a leak or fault with the system.';

   fact flowin :
      type : number;
      value :;
      status : param;
      question :;
      expln : 'The input flow';

   fact flowinternal :
      type : number;
      value :;
      status : param;
      question :;
      expln : 'The flow measured internally';

   fact flowout :
      type : number;
      value :;
```

```
      status : param;
      question :;
      expln : 'The flow coming out';

   fact leak :
      type : logical;
      value :;
      status : infer;
      question :;
      expln : 'Leak in flow pipe';

   fact leakbflow :
      type : logical;
      value :;
      status : infer;
      question :;
      expln : 'Leak before flow meter';

   fact leakaflow :
      type : logical;
      value :;
      status : infer;
      question :;
      expln : 'Leak after flow meter';

goal : if (leakbflow is t)  or (leakaflow is t) then leak =  t :
          'There is a leak is before or after the flow meter';
   1 : if (flowin > flowinternal)
          and (flowin >= flowout) then final leakbflow =  t :
          'There is a leak before the flow meter';
   2 : if (flowinternal > flowout)
          and (flowin >= flowout) then final leakaflow =  t :
          'There is a leak after the flow meter';

(* Partial rules *)

  10 : if (((flowin <> flowout) cor (flowout <> flowinternal))
             cor (flowin <> flowinternal))
             then partial noteqqual =  t :
             'The flows are not all equal';

  11 : if ((flowin < flowinternal) cor (flowinternal < flowout) ) cor
          (flowin < flowout)  then partial balance =  t :
         'The sensors are most likely faulty';


endmod;

pipe1 use pgfp (argument var flowin = mainflowin;
                         var flowinternal = mainflowinternal1;
                         val flowout = (mainflowinternal2 + maintflowout)) :
                         'Pipe 1';

tjunction use ptout (argument var flowin = mainflowinternal1;
                               var flowout = mainflowinternal2;
                               var tflowout = maintflowout;
                               var factor = factor) :
                               'Junction 1';

pipe2 use pgfp (argument val flowin = (mainflowinternal1 - maintflowout) ;
                         var flowinternal = mainflowinternal2;
                         var flowout = mainflowout) :
                         'Pipe 2';

goal : if ((pipe1) or (pipe2)) or (tjunction)
```

```
          then err =  t :
          'There is an fault in this line ';

endmod;

lineprob use line (argument val factor = 5) : 'Line 1';

goal : if lineprob then leakinline =  t :
          'The problem with the line is known';
endmod;
```

Linear Version

```
module main :

interface

module line :

partial : if (flowin <> (flowmeter2 + tflowout)) cor
             (flowmeter1 <> flowin) cor
             (flowmeter2 <> flowout) cor
             (tflowout <> (flowmeter1 / 2)) cor
             (flowout <> (flowmeter1 /2)) then partial perr =  t;

interface
   fact flowin :
      type : number;
      value :;
      status : ask;
      question :'What is the input flow';
      expln : 'The input flow';

   fact flowmeter1 :
      type : number;
      value :;
      status : ask;
      question :'What is the internal flow in pipe 1';
      expln : 'The flow measured internally on pipe 1';

   fact flowmeter2 :
      type : number;
      value :;
      status : ask;
      question :'What is the internal flow in pipe 2';
      expln : 'The flow measured internally on pipe 2';

   fact flowout :
      type : number;
      value :;
      status : ask;
      question :'What is the output flow';
      expln : 'The flow coming out';

   fact tflowout :
      type : number;
      value :;
      status : ask;
      question : 'What is the flow from the T-junction';
      expln : 'Flow from t-junction';
```

```
      fact leakbmeter1 :
         type : logical;
         status : infer;
         expln : 'Leak before meter 1';
      fact leakbmeter2 :
         type : logical;
         status : infer;
         expln : 'Leak before meter 2';
      fact leakameter1 :
         type : logical;
         status : infer;
         expln : 'Leak after meter 1';
      fact leakameter2 :
         type : logical;
         status : infer;
         expln : 'Leak after meter 2';
      fact blockedpipe :
         type : logical;
         status : infer;
         expln : 'Blocked pipe';
      fact blockedtpipe :
         type : logical;
         status : infer;
         expln : 'Blocked T-junction pipe';
      fact problem :
         type : logical;
         status : infer;
         expln : 'Problem with the pipe system';
      fact perr :
         type : logical;
         status : infer;
         expln : 'Problem with the line';

goal : if (leakbmeter1 is t) or
          (leakbmeter2 is t) or
          (leakameter1 is t) or
          (leakameter2 is t) or
          (blockedpipe is t) or
          (blockedtpipe is t)
       then problem =  t: 'There is a problem with the pipe';


   0 : if (flowmeter1 > (flowmeter2 + tflowout))
          then final  leakbmeter2 =  t :
          'Is there a leak in the pipe before meter 2';
   1 : if (flowmeter1 > (flowmeter2 + tflowout))
          then final  leakameter1 =  t :
          'Is there a leak after meter 1';
   2 : if (tflowout > (flowmeter1 / 2)) cand
          (flowin >= (flowout + tflowout))
          then final blockedpipe =  t:
          'If the T-flow out is greater than it is supposed to be';
   3 : if (flowmeter2 > (flowmeter1 - (flowmeter1 / 2))) cand
          (flowin >= (flowout + tflowout))
          then final blockedtpipe =  t:
          'If the flow out is greater than it should be';
   4 : if (flowin > flowmeter1) cand
          (flowin >= (flowout + tflowout))
          then final leakbmeter1 =  t :
          'There is a leak before the flow meter';
   5 : if (flowmeter2 > flowout) cand
          (flowin >= (flowout+tflowout))
          then final leakameter2 =  t :
          'There is a leak after the flow meter';
```

```
endmod;

fact prob :
   type : logical;
   status : infer;
   expln : 'A problem with the line';

lineprob use line : 'Line';

goal : if module lineprob then prob =  t :
       'The problem with the line is known';
endmod;
```

```
module main:

interface

fact sensor_fault :
     type : logical;
     status : infer;
     expln : 'There is a Faulty Sensor';

fact heat_problem :
     type : logical;
     status : infer;
     expln : 'There is a Problem with the Heaters';

fact steam_f :
     type : number;
     status : ask;
     question : 'What is the Steam Flow';
     expln : 'The Steam Flow';

fact steam_p :
     type : number;
     status : ask;
     question : 'What is the Steam Pressure';
     expln : 'The Steam Pressure';

fact steam_fv :
     type : number;
     status : ask;
     question : 'What is the Steam Flow Valve Opening';
     expln : 'The Steam Flow Valve Opening';

fact cs_t :
     type : number;
     status : ask;
     question : 'What is the Cold Solution Temperature';
     expln : 'Cold Solution Temperature';

fact shs_f :
     type : number;
     status : ask;
     question : 'What is the Cold Solution Flow';
     expln : 'Cold Solution Flow';

fact ft29 :
     type : number;
     status : ask;
     question : 'What is the Flow for Flow Meter 29';
     expln : 'Flow for Flow Meter 29';

fact ft30 :
     type : number;
     status : ask;
     question : 'What is the Flow for Flow Meter 30';
     expln : 'Flow for Flow Meter 30';

fact ft33_op :
     type : number;
     status : ask;
     question : 'What is Flow Valve Opening for Valve Number 33 ';
     expln : 'Flow Valve Opening for Valve Number 33 ';

fact steam_mf :
     type : number;
     status : ask;
```

```
     question : 'What is the Steam Mass Flow';
     expln : 'Steam Mass Flow';

fact ft33 :
     type : number;
     status : ask;
     question : 'What is the Flow for Flow Meter 33';
     expln : 'Flow for Flow Meter 33';


module heater_system (param flowin, flowout, steam_mf,
                      flowout_op, flowout_op_lo, flowout_op_hi):


interface

fact gen_fault :
     type : logical;
     status : infer;
     expln : 'Incorrectly controlling the heaters';

fact flowin :
     type : number;
     status : param;

fact flowout:
     type : number;
     status : param;

fact steam_mf:
     type : number;
     status : param;

fact flowout_op:
     type : number;
     status : param;

fact flowout_op_lo:
     type : number;
     status : param;

fact flowout_op_hi:
     type : number;
     status : param;

fact soln1in:
     type : number;
     status : infer;
     expln : 'Flow in to Heater 1';

fact soln1out:
     type : number;
     status : infer;
     expln : 'Flow out of Heater 1';

fact soln2in:
     type : number;
     status : infer;
     expln : 'Flow in to Heater 2';

fact soln2out:
     type : number;
     status : infer;
     expln : 'Flow out of Heater 2';
```

```
fact soln3in:
     type : number;
     status : infer;
     expln : 'Flow in to Heater 3';

fact soln3out:
     type : number;
     status : infer;
     expln : 'Flow out of Heater 3';

fact ft31a :
     type : number;
     status : ask;
     question : 'What is the value of Temperature Meter 31A';
     expln : 'Temperature Meter 31A';

fact ft31b :
     type : number;
     status : ask;
     question : 'What is the value of Temperature Meter 31B';
     expln : 'Temperature Meter 31B';

fact ft31c :
     type : number;
     status : ask;
     question : 'What is the value of Temperature Meter 31C';
     expln : 'Temperature Meter 31C';

fact ft32a :
     type : number;
     status : ask;
     question : 'What is the value of Temperature Meter 32A';
     expln : 'Temperature Meter 32A';

fact ft32b :
     type : number;
     status : ask;
     question : 'What is the value of Temperature Meter 32B';
     expln : 'Temperature Meter 32B';

fact ft32c :
     type : number;
     status : ask;
     question : 'What is the value of Temperature Meter 32C';
     expln : 'Temperature Meter 32C';

fact tempmid_sty_1:
     type : logical;
     status : ask;
     question : 'Is the Middle Temperature in Heater 1 Steady';
     expln : 'Middle Temperature in Heater 1 Steady';

fact tempout_sty_1:
     type : logical;
     status : ask;
     question : 'Is the Output Temperature in Heater 1 Steady';
     expln : 'Output Temperature in Heater 1 Steady';

fact tempmid_sty_2:
     type : logical;
     status : ask;
     question : 'Is the Middle Temperature in Heater 2 Steady';
     expln : 'Middle Temperature in Heater 2 Steady';

fact tempout_sty_2:
```

```
     type : logical;
     status : ask;
     question : 'Is the Output Temperature in Heater 2 Steady';
     expln : 'Output Temperature in Heater 2 Steady';

fact tempmid_sty_3:
     type : logical;
     status : ask;
     question : 'Is the Middle Temperature in Heater 3 Steady';
     expln : 'Middle Temperature in Heater 3 Steady';

fact tempout_sty_3:
     type : logical;
     status : ask;
     question : 'Is the Output Temperature in Heater 3 Steady';
     expln : 'Output Temperature in Heater 3 Steady';

fact select :
     type : number;
     value : 1..3;
     status : ask;
     question : 'Controlling from which heater  1 2 3 ';
     expln : 'Selected Heater';

fact heat_system :
     type : logical;
     status : infer;
     expln : 'There is some fault within the Heater System';


module heater (param flowin, flowout, tempmid, tempout, tempmid_sty,
                tempout_sty, select, steam_mf,
                flowout_op, flowout_op_lo, flowout_op_hi,
                steam_mf_lo, steam_mf_hi, heatno):

partial : if inconsistent is t then there_is_prob =  t;

interface

fact flowin :
     type : number;
     status : param;
fact flowout :
     type : number;
     status : param;
fact tempmid :
     type : number;
     status : param;
fact tempout :
     type : number;
     status : param;
fact tempmid_sty :
     type : logical;
     status : param;
fact tempout_sty :
     type : logical;
     status : param;
fact flowin :
     type : number;
     status : param;
fact select :
     type : number;
     status : param;
fact steam_mf :
     type : number;
```

```
     status : param;
fact flowout_op :
     type : number;
     status : param;
fact flowout_op_lo :
     type : number;
     status : param;
fact flowout_op_hi :
     type : number;
     status : param;
fact heatno :
     type : number;
     status : param;

fact leak :
     type : logical;
     status : infer;
     expln : 'LEAK in the HEATER System';

fact blockage :
     type : logical;
     status : infer;
     expln : 'BLOCKAGE in the HEATER System';

fact gen_fault :
     type : logical;
     status : infer;
     expln : 'Fault of a GENERAL Nature';

fact sensor_fault :
     type : logical;
     status : infer;
     expln : 'Faulty SENSOR';

fact heater_prob :
     type : logical;
     status : infer;
     expln : 'Problem with the Heater';

fact there_is_prob :
     type : logical;
     status : infer;
     expln : 'Problem with the Heater';

fact inconsistent :
     type : logical;
     status : infer;
     expln : 'Inconsistency in data';

fact leak_sensor :
     type : logical;
     status : infer;
     expln : 'Leak, blockage or Faulty Sensor';

fact temp_odd :
     type : logical;
     status : infer;
     expln : 'Temperature guages Faulty';

goal : if (leak is t) cor
          (blockage is t) cor
          (gen_fault is t) or
          (sensor_fault is t) then heater_prob =  t :
               'There is a fault with the heater';
```

```
   1 : if (select = heatno) cand (flowin > (flowout + 2)) cand
          (steam_mf < steam_mf_lo)
       then print 'There is a leak before or in the HEATER' &
            final leak =  t :' There is a leak before or in the HEATER';

   2 : if (select = heatno) cand (flowin > (flowout + 2))
          cand (steam_mf bt steam_mf_lo and steam_mf_hi) cand
               (flowout_op > flowout_op_hi)
       then print 'There is a leak after the HEATER' &
            final leak =  t :' There is a leak after the HEATER';

   3 : if (select = heatno) cand ((flowin - flowout) bt -2 and 2)
          cand (flowin < 2) cand
          (flowout_op > flowout_op_hi)
       then print 'There is a Blockage in the HEATER System' &
           final blockage =  t : 'There is a Blockage in the HEATER System';
   4 : if (tempout > 50) cand (select <> heatno)
       then print 'Controlling from the WRONG HEATER' &
            final gen_fault =  t : 'Controlling from the WRONG HEATER';

   5 : if (tempmid > 50) cand (select <> heatno)
       then print 'Controlling from the WRONG HEATER' &
            final gen_fault =  t : 'Controlling from the WRONG HEATER';

   6 : if (tempout <= 1)
       then print 'Faulty output temperature meters' &
            final sensor_fault =  t : 'Faulty output temperature meter';

   7 : if (tempmid <= 1)
       then print 'Faulty middle temperature meters' &
            final sensor_fault =  t : 'Faulty middle temperature meter';

   8 : if (tempout > 109)
       then print 'Faulty output temperature meters' &
            final sensor_fault =  t : 'Faulty output temperature meter';

   9 : if (tempmid > 109)
       then print 'Faulty middle temperature meters' &
            final sensor_fault =  t : 'Faulty middle temperature meter';

  10 : if (tempout > 50) cand (tempout_sty is t)
       then print 'Faulty Output temperature Guage' &
            final sensor_fault =  t :  'Faulty Output temperature Guage';

  11 : if (tempmid > 50) cand (tempmid_sty is t)
       then print 'Faulty Middle temperature Guage' &
            final sensor_fault =  t :  'Faulty Middle temperature Guage';

 100 : if ((flowin - flowout) nbt -2 and 2)
       then inconsistent =  t &
            partial leak_sensor =  t : 'There is a LEAK or FAULTY SENSOR';
 101 : if (tempmid > (tempout + 3))
       then inconsistent =  t &
            partial temp_odd =  t : 'TEMPOUT or TEMPMID is FAULTY';
endmod;



heater1 use heater ( argument   var flowin = soln1in;
                                var flowout = soln1out;
                                var tempmid = ft31a;
                                var tempout = ft32a;
                                var tempmid_sty = tempmid_sty_1;
                                var tempout_sty = tempout_sty_1;
```

```
                                  var select = select;
                                  var steam_mf = steam_mf;
                                  var flowout_op = flowout_op;
                                  val flowout_op_lo = 10;
                                  val flowout_op_hi = 50;
                                  val steam_mf_lo = 5;
                                  val steam_mf_hi = 15;
                                  val heatno = 1 ) : 'Heater 1';

heater2 use heater ( argument     var flowin = soln2in;
                                  var flowout = soln2out;
                                  var tempmid = ft31b;
                                  var tempout = ft32b;
                                  var tempmid_sty = tempmid_sty_2;
                                  var tempout_sty = tempout_sty_2;
                                  var select = select;
                                  var steam_mf = steam_mf;
                                  var flowout_op = flowout_op;
                                  val flowout_op_lo = 10;
                                  val flowout_op_hi = 50;
                                  val steam_mf_lo = 5;
                                  val steam_mf_hi = 15;
                                  val heatno = 2 ) : 'Heater 2';

heater3 use heater ( argument     var flowin = soln3in;
                                  var flowout = soln3out;
                                  var tempmid = ft31c;
                                  var tempout = ft32c;
                                  var tempmid_sty = tempmid_sty_3;
                                  var tempout_sty = tempout_sty_3;
                                  var select = select;
                                  var steam_mf = steam_mf;
                                  var flowout_op = flowout_op;
                                  val flowout_op_lo = 10;
                                  val flowout_op_hi = 50;
                                  val steam_mf_lo = 5;
                                  val steam_mf_hi = 15;
                                  val heatno = 3 ) : 'Heater 3';


goal : if (gen_fault is t) cor
          (heater1) cor
          (heater2) cor
          (heater3)
       then heat_system =  t  : 'There is a fault in the Heaters';

   1 : if (select <> 1) cand (steam_mf > 1) cand
          ((ft31a > 50) or (ft32a > 50))
       then print 'Incorrectly Controlling from Heater 1' &
            gen_fault =  t : 'Incorrectly Controlling from Heater 1';

   2 : if (select <> 2) cand (steam_mf > 1) cand
          ((ft31b > 50) or (ft32b > 50))
       then print 'Incorrectly Controlling from Heater 2' &
            gen_fault =  t : 'Incorrectly Controlling from Heater 2';

   3 : if (select <> 3) cand (steam_mf > 1) cand
          ((ft31c > 50) or (ft32c > 50))
       then print 'Incorrectly Controlling from Heater 3' &
            gen_fault =  t : 'Incorrectly Controlling from Heater 3';

   4: if select = 1 then soln1in =  flowin & soln1out =  flowout  &
                         soln2in =  0 & soln2out =  0 &
                         soln3in =  0 & soln3out =  0 :
                         'Setting the flows';
```

```
   5: if select = 2 then soln2in =  flowin & soln2out =  flowout  &
                         soln1in =  0 & soln1out =  0 &
                         soln3in =  0 & soln3out =  0 :
                         'Setting the flows';

   6: if select = 3 then soln3in =  flowin & soln3out =  flowout  &
                         soln2in =  0 & soln2out =  0 &
                         soln1in =  0 & soln1out =  0 :
                         'Setting the Flows';
endmod;


solnheater use heater_system ( argument val fiowin = ft29 + ft30;
                                       val flowout = ft33;
                                       var steam_mf = steam_mf;
                                       var flowout_op = ft33_op;
                                       val flowout_op_lo = 20;
                                       val flowout_op_hi = 80) : 'Heater System';

goal : if (solnheater) cor
          (sensor_fault is t)
       then heat_problem =  t : 'Problem with in the Heater System';

   1 : if  (steam_f <= 0) and
           (steam_fv <> 0)
       then sensor_fault =  t &
            print 'Faulty Steam Flow Meter' :
            'Faulty Steam Flow Meter';

   2 : if  (steam_f >= 2600)
       then sensor_fault =  t &
            print 'Faulty Steam Flow Meter' :
            'Faulty Steam Flow Meter';

   3 : if  (steam_p <= 0)  and
           (steam_fv <> 0)
       then sensor_fault =  t &
            print 'Faulty Steam Pressure Meter' :
            'Faulty Steam Pressure Meter';

   4 : if  (steam_p >= 350)
       then sensor_fault =  t &
            print 'Faulty Steam Pressure Meter' :
            'Faulty Steam Pressure Meter';

   5 : if cs_t <= 0
       then sensor_fault =  t &
            print 'Faulty Input Temperature' :
                  'Faulty Input Temperature';

   6 : if cs_t >= 110
       then sensor_fault =  t &
            print 'Faulty Input Temperature' :
                  'Faulty Input Temperature';

   7 : if (shs_f <= 0)
       then sensor_fault =  t &
            print 'Faulty SHS-F' :
            'Faulty SHS-F';

   8 : if (shs_f >= 250)
       then sensor_fault =  t &
            print 'Faulty SHS-F' :
            'Faulty SHS-F';
```

```
endmod;
```

```
module main:

interface

fact leak :
     type : logical;
     status : infer;
     expln : 'There is a Leak';

fact blockage:
     type : logical;
     status : infer;
     expln : 'There is a Blockage';

fact gen_fault :
     type : logical;
     status : infer;
     expln : 'There is a Fault';

fact sensor_fault :
     type : logical;
     status : infer;
     expln : 'There is a Faulty Sensor';

fact heater_problem :
     type : logical;
     status : infer;
     expln : 'There is a Problem with the Heaters';

fact steam_f :
     type : number;
     status : ask;
     question : 'What is the Steam Flow';
     expln : 'The Steam Flow';

fact steam_fv :
     type : number;
     status : ask;
     question : 'What is the Steam Flow Valve Opening';
     expln : 'The Steam Flow Valve Opening';

fact cs_t :
     type : number;
     status : ask;
     question : 'What is the Cold Solution Temperature';
     expln : 'Cold Solution Temperature';

fact shs_f :
     type : number;
     status : ask;
     question : 'What is the Cold Solution Flow';
     expln : 'Cold Solution Flow';

fact ft29 :
     type : number;
     status : ask;
     question : 'What is the Flow for Flow Meter 29';
     expln : 'Flow for Flow Meter 29';

fact ft30 :
     type : number;
     status : ask;
     question : 'What is the Flow for Flow Meter 30';
     expln : 'Flow for Flow Meter 30';
```

```
fact ft33_op :
     type : number;
     status : ask;
     question : 'What is Flow Valve Opening for Valve Number 33 ';
     expln : 'Flow Valve Opening for Valve Number 33 ';

fact steam_mf :
     type : number;
     status : ask;
     question : 'What is the Steam Mass Flow';
     expln : 'Steam Mass Flow';

fact steam_mf_lo :
     type : number;
     status : ask;
     question : 'What is the Steam Mass Flow LOW';
     expln : 'Steam Mass Flow LOW';

fact steam_mf_hi :
     type : number;
     status : ask;
     question : 'What is the Steam Mass Flow HIGH';
     expln : 'Steam Mass Flow HIGH';

fact ft33 :
     type : number;
     status : ask;
     question : 'What is the Flow for Flow Meter 33';
     expln : 'Flow for Flow Meter 33';



fact soln1in:
     type : number;
     status : infer;
     expln : 'Flow in to Heater 1';

fact soln1out:
     type : number;
     status : infer;
     expln : 'Flow out of Heater 1';

fact soln2in:
     type : number;
     status : infer;
     expln : 'Flow in to Heater 2';

fact soln2out:
     type : number;
     status : infer;
     expln : 'Flow out of Heater 2';

fact soln3in:
     type : number;
     status : infer;
     expln : 'Flow in to Heater 3';

fact soln3out:
     type : number;
     status : infer;
     expln : 'Flow out of Heater 3';

fact ft31a :
     type : number;
```

```
     status : ask;
     question : 'What is the value of Temperature Meter 31A';
     expln : 'Temperature Meter 31A';

fact ft31b :
     type : number;
     status : ask;
     question : 'What is the value of Temperature Meter 31B';
     expln : 'Temperature Meter 31B';

fact ft31c :
     type : number;
     status : ask;
     question : 'What is the value of Temperature Meter 31C';
     expln : 'Temperature Meter 31C';

fact ft32a :
     type : number;
     status : ask;
     question : 'What is the value of Temperature Meter 32A';
     expln : 'Temperature Meter 32A';

fact ft32b :
     type : number;
     status : ask;
     question : 'What is the value of Temperature Meter 32B';
     expln : 'Temperature Meter 32B';

fact ft32c :
     type : number;
     status : ask;
     question : 'What is the value of Temperature Meter 32C';
     expln : 'Temperature Meter 32C';

fact tempmid_sty_1:
     type : logical;
     status : ask;
     question : 'Is the Middle Temperature in Heater 1 Steady';
     expln : 'Middle Temperature in Heater 1 Steady';

fact tempout_sty_1:
     type : logical;
     status : ask;
     question : 'Is the Output Temperature in Heater 1 Steady';
     expln : 'Output Temperature in Heater 1 Steady';

fact tempmid_sty_2:
     type : logical;
     status : ask;
     question : 'Is the Middle Temperature in Heater 2 Steady';
     expln : 'Middle Temperature in Heater 2 Steady';

fact tempout_sty_2:
     type : logical;
     status : ask;
     question : 'Is the Output Temperature in Heater 2 Steady';
     expln : 'Output Temperature in Heater 2 Steady';

fact tempmid_sty_3:
     type : logical;
     status : ask;
     question : 'Is the Middle Temperature in Heater 3 Steady';
     expln : 'Middle Temperature in Heater 3 Steady';

fact tempout_sty_3:
```

```
     type : logical;
     status : ask;
     question : 'Is the Output Temperature in Heater 3 Steady';
     expln : 'Output Temperature in Heater 3 Steady';

fact select :
     type : number;
     value : 1..3;
     status : ask;
     question : 'Controlling from which heater  1 2 3 ';
     expln : 'Selected Heater';


fact inconsistent :
     type : logical;
     status : infer;
     expln : 'Inconsistency in data';

fact leak_sensor :
     type : logical;
     status : infer;
     question : 'Leak, blockage or Faulty Sensor';

fact temp_odd :
     type : logical;
     status : infer;
     question : 'Temperature guages Faulty';


goal : if (leak is t) cor
          (blockage is t) cor
          (gen_fault is t) or
          (sensor_fault is t) then heater_prob =  t :
               'There is a fault with the heater';

   21 : if (select = 1) cand (soln1in > (soln1out + 2)) cand
          (steam_mf < steam_mf_lo)
        then print 'There is a leak before or in the HEATER' &
             final leak =  t :' There is a leak before or in the HEATER';

   22 : if (select = 1) cand (soln1in > (soln1out + 2))
          cand (steam_mf bt steam_mf_lo and steam_mf_hi) cand
               (soln1out_op > soln1out_op_hi)
        then print 'There is a leak after the HEATER' &
             final leak =  t :' There is a leak after the HEATER';

   23 : if (select = 1) cand ((soln1in - soln1out) bt -2 and 2)
          cand (soln1in < soln1in_lo) cand
          (soln1out_op > soln1out_op_hi)
        then print 'There is a Blockage in the HEATER System' &
            final blockage =  t : 'There is a Blockage in the HEATER System';
   24 : if (ft32a > 50) cand (select <> 1)
        then print 'Controlling from the WRONG HEATER' &
             final gen_fault =  t : 'Controlling from the WRONG HEATER';

   25 : if (ft31a > 50) cand (select <> 1)
        then print 'Controlling from the WRONG HEATER' &
             final gen_fault =  t : 'Controlling from the WRONG HEATER';

   26 : if (ft32a <= 1)
        then print 'Faulty output temperature meters' &
             final sensor_fault =  t : 'Faulty output temperature meter';

   27 : if (ft31a <= 1)
        then print 'Faulty middle temperature meters' &
```

```
          final sensor_fault =  t : 'Faulty middle temperature meter';

   28 : if (ft32a > 109)
        then print 'Faulty output temperature meters' &
             final sensor_fault =  t : 'Faulty output temperature meter';

   29 : if (ft31a > 109)
        then print 'Faulty middle temperature meters' &
             final sensor_fault =  t : 'Faulty middle temperature meter';

  210 : if (ft32a > 50) cand (tempout_sty_1 is t)
        then print 'Faulty Output temperature Guage' &
             final sensor_fault =  t :  'Faulty Output temperature Guage';

  211 : if (ft31a > 50) cand (tempmid_sty_1 is t)
        then print 'Faulty Middle temperature Guage' &
             final sensor_fault =  t :  'Faulty Middle temperature Guage';

 2100 : if ((soln1in - soln1out) nbt -2 and 2)
        then inconsistent =  t &
             partial leak_sensor =  t : 'There is a LEAK or FAULTY SENSOR';

 2101 : if (ft31a > (ft32a + 3))
        then inconsistent =  t &
             partial temp_odd =  t : 'TEMPOUT or TEMPMID is FAULTY';

   31 : if (select = 2) cand (soln2in > (soln2out + 2)) cand
           (steam_mf < steam_mf_lo)
        then print 'There is a leak before or in the HEATER' &
             final leak =  t :' There is a leak before or in the HEATER';

   32 : if (select = 2) cand (soln2in > (soln2out + 2))
          cand (steam_mf bt steam_mf_lo and steam_mf_hi) cand
                (soln2out_op > soln2out_op_hi)
        then print 'There is a leak after the HEATER' &
             final leak =  t :' There is a leak after the HEATER';

   33 : if (select = 2) cand ((soln2in - soln2out) bt -2 and 2)
          cand (soln2in < soln2in_lo) cand
          (soln2out_op > soln2out_op_hi)
        then print 'There is a Blockage in the HEATER System' &
            final blockage =  t : 'There is a Blockage in the HEATER System';
   34 : if (ft32b > 50) cand (select <> 2)
        then print 'Controlling from the WRONG HEATER' &
             final gen_fault =  t : 'Controlling from the WRONG HEATER';

   35 : if (ft31b > 50) cand (select <> 2)
        then print 'Controlling from the WRONG HEATER' &
             final gen_fault =  t : 'Controlling from the WRONG HEATER';

   36 : if (ft32b <= 1)
        then print 'Faulty output temperature meters' &
             final sensor_fault =  t : 'Faulty output temperature meter';

   37 : if (ft31b <= 1)
        then print 'Faulty middle temperature meters' &
             final sensor_fault =  t : 'Faulty middle temperature meter';

   38 : if (ft32b > 109)
        then print 'Faulty output temperature meters' &
             final sensor_fault =  t : 'Faulty output temperature meter';

   39 : if (ft31b > 109)
        then print 'Faulty middle temperature meters' &
             final sensor_fault =  t : 'Faulty middle temperature meter';
```

```
 310 : if (ft32b > 50) cand (tempout_sty_2 is t)
      then print 'Faulty Output temperature Guage' &
           final sensor_fault =  t :  'Faulty Output temperature Guage';

 311 : if (ft31b > 50) cand (tempmid_sty_2 is t)
      then print 'Faulty Middle temperature Guage' &
           final sensor_fault =  t :  'Faulty Middle temperature Guage';

3100 : if ((soln2in - soln2out) nbt -2 and 2)
      then inconsistent =  t &
           partial leak_sensor =  t : 'There is a LEAK or FAULTY SENSOR';

3101 : if (ft31b > (ft32b + 3))
      then inconsistent =  t &
           partial temp_odd =  t : 'TEMPOUT or TEMPMID is FAULTY';

  41 : if (select = 3) cand (soln3in > (soln3out + 2)) cand
         (steam_mf < steam_mf_lo)
      then print 'There is a leak before or in the HEATER' &
           final leak =  t :' There is a leak before or in the HEATER';

  42 : if (select = 3) cand (soln3in > (soln3out + 2))
        cand (steam_mf bt steam_mf_lo and steam_mf_hi) cand
              (soln3out_op > soln3out_op_hi)
      then print 'There is a leak after the HEATER' &
           final leak =  t :' There is a leak after the HEATER';

  43 : if (select = 3) cand ((soln3in - soln3out) bt -2 and 2)
        cand (soln3in < soln3in_lo) cand
        (soln3out_op > soln3out_op_hi)
      then print 'There is a Blockage in the HEATER System' &
          final blockage =  t : 'There is a Blockage in the HEATER System';
  44 : if (ft32c > 50) cand (select <> 3)
      then print 'Controlling from the WRONG HEATER' &
           final gen_fault =  t : 'Controlling from the WRONG HEATER';

  45 : if (ft31c > 50) cand (select <> 3)
      then print 'Controlling from the WRONG HEATER' &
           final gen_fault =  t : 'Controlling from the WRONG HEATER';

  46 : if (ft32c <= 1)
      then print 'Faulty output temperature meters' &
           final sensor_fault =  t : 'Faulty output temperature meter';

  47 : if (ft31c <= 1)
      then print 'Faulty middle temperature meters' &
           final sensor_fault =  t : 'Faulty middle temperature meter';

  48 : if (ft32c > 109)
      then print 'Faulty output temperature meters' &
           final sensor_fault =  t : 'Faulty output temperature meter';

  49 : if (ft31c > 109)
      then print 'Faulty middle temperature meters' &
           final sensor_fault =  t : 'Faulty middle temperature meter';

 410 : if (ft32c > 50) cand (tempout_sty_3 is t)
      then print 'Faulty Output temperature Guage' &
           final sensor_fault =  t :  'Faulty Output temperature Guage';

 411 : if (ft31c > 50) cand (tempmid_sty_3 is t)
      then print 'Faulty Middle temperature Guage' &
           final sensor_fault =  t :  'Faulty Middle temperature Guage';
```

```
4100 : if ((soln3in - soln3out) nbt -2 and 2)
     then inconsistent =  t &
          partial leak_sensor =  t : 'There is a LEAK or FAULTY SENSOR';
4101 : if (ft31c > (ft32c + 3))
     then inconsistent =  t &
          partial temp_odd =  t : 'TEMPOUT or TEMPMID is FAULTY';

  11 : if (select <> 1) cand (steam_mf > 1) cand
         ((ft31a > 50) or (ft32a > 50))
     then print 'Incorrectly Controlling from Heater 1' &
          gen_fault =  t : 'Incorrectly Controlling from Heater 1';

  12 : if (select <> 2) cand (steam_mf > 1) cand
         ((ft31b > 50) or (ft32b > 50))
     then print 'Incorrectly Controlling from Heater 2' &
          gen_fault =  t : 'Incorrectly Controlling from Heater 2';

  13 : if (select <> 3) cand (steam_mf > 1) cand
         ((ft31c > 50) or (ft32c > 50))
     then print 'Incorrectly Controlling from Heater 3' &
          gen_fault =  t : 'Incorrectly Controlling from Heater 3';

  14: if select = 1 then soln1in =  (ft29 + ft30) & soln1out =  ft33  &
                         soln2in =  0 & soln2out =  0 &
                         soln3in =  0 & soln3out =  0 :
                         'Setting the flows';

  15: if select = 2 then soln2in =  (ft29 + ft30) & soln2out =  ft33  &
                         soln1in =  0 & soln1out =  0 &
                         soln3in =  0 & soln3out =  0 :
                         'Setting the flows';

  16: if select = 3 then soln3in =  (ft29 + ft30) & soln3out =  ft33  &
                         soln2in =  0 & soln2out =  0 &
                         soln1in =  0 & soln1out =  0 :
                         'Setting the Flows';



  1 : if  (steam_f <= 0) and
          (steam_fv <> 0)
     then sensor_fault =  t &
          print 'Faulty Steam Flow Meter' :
          'Faulty Steam Flow Meter';

  2 : if  (steam_f >= 2600)
     then sensor_fault =  t &
          print 'Faulty Steam Flow Meter' :
          'Faulty Steam Flow Meter';

  3 : if  (steam_p <= 0)  and
          (steam_fv <> 0)
     then sensor_fault =  t &
          print 'Faulty Steam Pressure Meter' :
          'Faulty Steam Pressure Meter';

  4 : if  (steam_p >= 350)
     then sensor_fault =  t &
          print 'Faulty Steam Pressure Meter' :
          'Faulty Steam Pressure Meter';

  5 : if cs_t <= 0
     then sensor_fault =  t &
          print 'Faulty Input Temperature' :
                'Faulty Input Temperature';
```

```
   6 : if cs_t >= 110
       then sensor_fault =  t &
            print 'Faulty Input Temperature' :
                  'Faulty Input Temperature';

   7 : if (shs_f <= 0)
       then sensor_fault =  t &
            print 'Faulty SHS-F' :
            'Faulty SHS-F';

   8 : if (shs_f >= 250)
       then sensor_fault =  t &
            print 'Faulty SHS-F' :
            'Faulty SHS-F';

endmod;
```

Table 1 Modular version
Number of references to facts and parameters and the number of rules where a fact can appear. (Appendix G)

Facts Name	No Refs	No Rules
Module Main		
sensor_fault	11	11
heat_problem	1	11
steam_f	2	11
steam_fv	1	11
cs_t	2	11
shs_f	2	11
shs_p	2	11
ft29	1	11
ft30	1	11
ft33	4	18
ft33_op	6	32
steam_mf	12	32
Module Heater		
leak	3	14
blockage	2	14
gen_fault	2	14
sensor_fault	7	14
heater_prob	1	14
there_is_prob	1	14
inconsistent	3	14
leak_sensor	1	14
temp_odd	1	14
Module Heater_System		
flowin	3	7
soln1in	8	21
soln1out	8	21
soln2in	8	21
soln2out	8	21
soln3in	8	21
soln3out	8	21
ft31a	8	21
ft31b	8	21
ft31c	8	21
ft32a	8	21
ft32b	8	21
ft32c	8	21
tempmid_sty_1	2	21
tempmid_sty_2	2	21
tempmid_sty_3	2	21
tempout_sty_1	2	21
tempout_sty_2	2	21
tempout_sty_3	2	21
select	14	21
gen_fault	5	7
heat_system	1	7

Table 1 Modular version
Number of references to facts and parameters and the number of rules where a fact can appear. (Appendix E)

Facts Name	No Refs	No Rules
Module Main		
leakinline	1	1
Module Line		
factor	4	5
err	1	2
mainflowin	3	4
mainflowinternal1	10	7
mainflowinternal2	6	7
mainflowout	4	4
maintflowout	4	5
Module PGFP		
leak	1	3
keakbflow	2	3
leakaflow	2	3
Module PTOUT		
leak	2	4
blockedpipe	2	4
blockedtpipe	2	4
err	1	4

Table 2 Linear version
Number of references to facts and the number of rules where a fact can appear.

Facts Name	No Refs	No Rules
flowin	4	7
flowmeter1	5	7
flowmeter2	4	7
flowout	4	7
tflowout	6	7
leakbmeter1	2	7
leakbmeter2	2	7
leakameter1	2	7
leakameter2	2	7
blockedpipe	2	7
blockedtpipe	2	7
problem	1	7

```
module main :

interface

fact front_ok:
     type : logical;
     status : infer;
     expln : 'All the front Lights work';

fact back_ok:
     type : logical;
     status : infer;
     expln : 'All the back Lights work';

fact all_ok:
     type : logical;
     status : infer;
     expln : 'All the Lights work';

fact head_work:
     type : logical;
     status : ask;
     question : 'Do the Head Lights work when turned on';
     expln    : 'The Head Lights works when turned on';

fact head_alligned:
     type : logical;
     status : ask;
     question : 'Are the Head Lights Alligned properly';
     expln    : 'The Head Lights Alligned properly';

fact spots_work:
     type : logical;
     status : ask;
     question : 'Does the Spot Light work when turned on';
     expln    : 'The Spot Light works when turned on';

fact spots_alligned:
     type : logical;
     status : ask;
     question : 'Is the Spot Light alligned properly';
     expln    : 'The Spot Light is alligned properly';

fact spots_ok:
     type : logical;
     status : infer;
     expln : 'The Spot Lights are alligned and work properly';

fact head_ok :
     type : logical;
     status : infer;
     expln : 'The Head Lights are alligned and work properly';

fact tail_work:
     type : logical;
     status : ask;
     question : 'Do the tail lights work';
     expln : 'The Tail lights work properly';

fact stop_work:
     type : logical;
     status : ask;
     question : 'Does the Stop Light work properly';
     expln : 'The Stop Light works properly';
```

```
fact tail_ok:
     type : logical;
     status : infer;
     expln : 'The Tail Lights work properly';

fact hi_stop_ok:
     type : logical;
     status : infer;
     expln : 'The High Stop Light works properly';

fact hi_stop_work:
     type : logical;
     status : ask;
     question : 'Does the High Stop Light work properly';
     expln : 'The High Stop Light works properly';



goal : if (front_ok is t)   cand (back_ok is t) then final all_ok = t :
          'Check if the Front and Back Lights work';

1:       if (head_ok is t) cand (spots_ok is t)
         then front_ok = t :
           'Check the Head lights and Spot Lights if they exist';

2:       if head_work is t and head_alligned is t then head_ok = t :
           'Check if the headlights work and are alligned';

3:       if (spots_work is t) cand (spots_alligned is t) then spots_ok = t :
           'Check if the spotlights work and alligned';

4:       if spots_work is i then spots_ok = t :
           'Spot_lights do not exits';

5:       if tail_ok is t cand ((hi_stop_ok is t) cor (hi_stop_ok is i))
         then back_ok = t :
           'Check the Head lights and Spot Lights if they exist';

6:       if tail_work is t and stop_work is t then tail_ok = t :
           'Check if the taillights work and are alligned';

7:       if hi_stop_work is t then hi_stop_ok = t :
           'Check if the spotlights work and alligned';

8:       if hi_stop_work is i then hi_stop_ok = i :
           'Spot_lights do not exits';

endmod;
```

```
module main :

interface

fact front_ok:
     type : logical;
     status : infer;
     expln : 'All the front Lights work';

fact back_ok:
     type : logical;
     status : infer;
     expln : 'All the back Lights work';

fact all_ok:
     type : logical;
     status : infer;
     expln : 'All the Lights work';

fact head_work:
     type : logical;
     status : ask;
     question : 'Do the Head Lights work when turned on';
     expln    : 'The Head Lights works when turned on';

fact head_alligned:
     type : logical;
     status : ask;
     question : 'Are the Head Lights Alligned properly';
     expln    : 'The Head Lights Alligned properly';

fact spots_work:
     type : logical;
     status : ask;
     question : 'Does the Spot Light work when turned on';
     expln    : 'The Spot Light works when turned on';

fact have_spots:
     type : logical;
     status : ask;
     question : 'Does the car have Spot Lights';
     expln    : 'The car has Spot Lights';

fact spots_alligned:
     type : logical;
     status : ask;
     question : 'Is the Spot Light alligned properly';
     expln    : 'The Spot Light is alligned properly';

fact spots_ok:
     type : logical;
     status : infer;
     expln : 'The Spot Lights are alligned and work properly';

fact head_ok :
     type : logical;
     status : infer;
     expln : 'The Head Lights are alligned and work properly';

fact tail_work:
     type : logical;
     status : ask;
     question : 'Do the tail lights work';
     expln : 'The Tail lights work properly';
```

```
fact stop_work:
     type : logical;
     status : ask;
     question : 'Does the Stop Light work properly';
     expln : 'The Stop Light works properly';

fact tail_ok:
     type : logical;
     status : infer;
     expln : 'The Tail Lights work properly';

fact hi_stop_ok:
     type : logical;
     status : infer;
     expln : 'The High Stop Light works properly';

fact hi_stop_work:
     type : logical;
     status : ask;
     question : 'Does the High Stop Light work properly';
     expln : 'The High Stop Light works properly';

fact have_hi_stop:
     type : logical;
     status : ask;
     question : 'Does the car have High Stop Lighs';
     expln : 'The car has High Stop Lights';



goal : if (front_ok is t)   cand (back_ok is t) then final all_ok = t :
          'Check if the Front and Back Lights work';

1:       if (head_ok is t) cand (spots_ok is t)
         then front_ok = t :
           'Check the Head lights and Spot Lights if they exist';

2:       if head_work is t and head_alligned is t then head_ok = t :
            'Check if the headlights work and are alligned';

3:       if (have_spots is t) cand ((spots_work is t) cand (spots_alligned is t))
             then spots_ok = t :
            'Check if the spotlights work and alligned';

4:       if have_spots is f then spots_ok = t :
            'Spot_lights do not exits';

5:       if tail_ok is t cand ((hi_stop_ok is t) cor (hi_stop_ok is i))
         then back_ok = t :
           'Check the Head lights and Spot Lights if they exist';

6:       if tail_work is t and stop_work is t then tail_ok = t :
            'Check if the taillights work and are alligned';

7:       if (have_hi_stop is t) cand (hi_stop_work is t_) then hi_stop_ok = t :
            'Check if the spotlights work and alligned';

8:       if have_hi_spot is f then hi_stop_ok = i :
            'Spot_lights do not exits';

endmod;
```

```
or(t,t,t).
or(t,f,t).
or(t,d,t).
or(t,u,t).
or(t,i,t).
or(f,t,t).
or(f,f,f).
or(f,d,d).
or(f,u,u).
or(f,i,i).
or(d,t,t).
or(d,f,d).
or(d,d,d).
or(d,u,u).
or(d,i,u).
or(u,t,t).
or(u,f,u).
or(u,d,u).
or(u,u,u).
or(u,i,u).
or(i,t,t).
or(i,f,i).
or(i,d,u).
or(i,u,u).
or(i,i,i).

and(t,t,t).
and(t,f,f).
and(t,d,d).
and(t,u,u).
and(t,i,u).
and(f,t,f).
and(f,f,f).
and(f,d,f).
and(f,u,f).
and(f,i,f).
and(d,t,d).
and(d,f,f).
and(d,d,d).
and(d,u,u).
and(d,i,u).
and(u,t,u).
and(u,f,f).
and(u,d,u).
and(u,u,u).
and(u,i,u).
and(i,t,i).
and(i,f,f).
and(i,d,u).
and(i,u,u).
and(i,i,i).

neg(t,f).
neg(f,t).
neg(d,i).
neg(u,u).
neg(i,d).



imp(X,Y,Z) :- neg(X,X1), or(X1,Y,Z).

equ(X,Y,Z) :- imp(X,Y,U), imp(Y,X,V), and(U,V,Z).
```

```
/* Short hand rules */

and(X,Y,Z,R) :- and(X,Y,R1),and(R1,Z,R).
and(W,X,Y,Z,R) :- and(W,X,R1),and(R1,Y,R2),and(R2,Z,R).
and(V,W,X,Y,Z,R) :- and(V,W,R1),and(R1,X,R2),and(R2,Y,R3),and(R3,Z,R).

or(X,Y,Z,R) :- or(X,Y,R1),or(R1,Z,R).
or(W,X,Y,Z,R) :- or(W,X,R1),or(R1,Y,R2),or(R2,Z,R).
or(V,W,X,Y,Z,R) :- or(V,W,R1),or(R1,X,R2),or(R2,Y,R3),or(R3,Z,R).
```

```
In Heater System Controlling from which heater  1 2 3 ?
Values : 1..3
Answer : 1.
In The Main Module What is the Steam Mass Flow?
Values : Any number
Answer : 1.
In The Main Module What is the Flow for Flow Meter 29?
Values : Any number
Answer : 25.
In The Main Module What is the Flow for Flow Meter 30?
Values : Any number
Answer : 5.
In The Main Module What is the Flow for Flow Meter 33?
Values : Any number
Answer : 1.
There is a leak before or in the HEATER
In Heater System What is the value of Temperature Meter 32A?
Values : Any number
Answer : 95.
In Heater System What is the value of Temperature Meter 31A?
Values : Any number
Answer : 90.
In Heater System Is the Output Temperature in Heater 1 Steady?
Values : t f d u i
Answer : f.
In Heater System Is the Middle Temperature in Heater 1 Steady?
Values : t f d u i
Answer : f.
Result : t
In Heater 1, LEAK in the HEATER System has the value True.
 also On a more general level,
In Heater 1, Leak, blockage or Faulty Sensor has the value True,
nothing else is known.
```